Учебник для вузов

ПИТЕР

СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

С. А. Орлов

Теория и практика языков программирования

ДЛЯ БАКАЛАВРОВ И МАГИСТРОВ

РЕКОМЕНДОВАНО РОССИЙСКОЙ АКАДЕМИЕЙ НАУК

Сергей Александрович Орлов

Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения

Рецензенты:

Соколов Б. В., д. т. н., профессор, руководитель лаборатории информационных технологий в системном анализе и моделировании Санкт-Петербургского института информатики и автоматизации РАН

Пасмуров А. Я., д. т. н., доцент, senior member of IEEE, руководитель учебного центра ФГУП «ЗащитаИнфоТранс» Министерства транспорта РФ, Санкт-Петербургский филиал

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Л. Родионова*

ББК 32.973.2-018.1я7
УДК 004.43(075)

Орлов С. А.

О-66 Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения. — СПб.: Питер, 2013. — 688 с.: ил.

ISBN 978-5-496-00032-1

Учебник посвящен систематическому изложению теории и практики языков программирования. Он отражает классическое содержание учебной дисциплины по языкам программирования. Все сложные вопросы поясняются законченными примерами. Кроме того, здесь предлагается полный комплекс задач и упражнений по узловым вопросам. В российских стандартах третьего поколения (компьютерного направления) учебник охватывает базисные разделы следующих дисциплин: теория формальных языков, теория автоматов и формальных языков, языки программирования, программирование, объектно-ориентированное программирование, логическое и функциональное программирование, теория вычислительных процессов.

Учебник предназначен для студентов инженерного, бакалаврского и магистерского уровней компьютерных специальностей, может быть полезен преподавателям и исследователям/разработчикам трансляторов и другого программного обеспечения.

Рекомендовано Санкт-Петербургским институтом информатики и автоматизации Российской академии наук (СПИИРАН) в качестве учебника по направлению «Информатика и вычислительная техника».

ISBN 978-5-496-00032-1

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 11.10.12. Формат 70x100/16. Усл. п. л. 55,470. Тираж 2000. Заказ
Отпечатано с готовых диапозитивов в ИПК ООО «Ленинградское издательство».
194044, Санкт-Петербург, ул. Менделеевская, 9.

Оглавление

Введение

Шарлатан тот, кто не может простыми
словами объяснить, чем занимается.

К. Воннегут

Известно, что основной задачей первых трех десятилетий компьютерной эры являлось развитие аппаратных компьютерных средств. Это было обусловлено высокой стоимостью обработки и хранения данных. В 80-е годы успехи микроэлектроники привели к резкому увеличению производительности компьютера при значительном снижении стоимости.

Основной задачей начала XXI века стало совершенствование качества компьютерных систем, возможности которых целиком определяются программным обеспечением (ПО).

Современный персональный компьютер теперь имеет производительность большой ЭВМ 80-х годов. Сняты практически все аппаратные ограничения на решение задач. Оставшиеся ограничения приходятся на долю ПО.

Чрезвычайно актуальными стали следующие проблемы:

- аппаратная сложность опережает наше умение строить ПО, использующее потенциальные возможности аппаратуры;
- наше умение строить новые программы отстает от требований к новым программам;
- нашим возможностям эксплуатировать существующие программы угрожает низкое качество их разработки.

Ключом к решению этих проблем является понимание и грамотное использование языка программирования как основного инструмента для создания ПО, разумная организация процесса разработки программного приложения.

Настоящий учебник посвящен систематическому изложению теории и практики всего спектра современных языков программирования: принципов их построения, оценки и использования.

В основу материала положен многолетний опыт постановки и преподавания автором дисциплин по языкам программирования. Базовый курс «Принципы построения языков программирования» прослушали больше двух тысяч студентов, работающих теперь в инфраструктуре мировой программной индустрии самых разных стран и на самых разных континентах.

Автор стремился к достижению трех целей:

- изложить классические основы, отражающие накопленный мировой опыт в области языков программирования;
- показать научные и практические достижения, характеризующие динамику развития языков программирования;

- обобщить и отразить 20-летний университетский опыт постановки и преподавания соответствующих дисциплин.

Компьютерные науки — очень популярные и стремительно развивающиеся области знаний. Обоснование простое: человеческое общество XXI века — информационное общество. Об этом говорят цифры: в ведущих странах занятость населения в информационной сфере составляет 60%, а в сфере материального производства — 40%. Именно поэтому специальности направления «Компьютерные науки и информационные технологии» гарантируют приобретение наиболее престижных, дефицитных и высокооплачиваемых профессий. Так считают во всех развитых странах мира. Ведь не зря утверждают: «Кто владеет информацией — тот владеет миром!»

Поэтому понятно то пристальное внимание, которое уделяет компьютерному образованию мировое сообщество, понятно стремление унифицировать и упорядочить знания, необходимые специалисту этого направления. Одним из результатов такой работы является международный стандарт по компьютерному образованию Computing Curricula 2005, состоящий из пяти томов (отметим том Computer Science Curriculum 2008).

Содержание данного учебника отвечает рекомендациям этого стандарта. Учебник состоит из 21 главы.

Первая глава посвящена базовым понятиям языков программирования. Здесь вводятся определения языка с трех точек зрения: семиотической, практической и технологической, поясняется предлагаемый подход к изучению (принцип абстракции-конкретизации), дается обзорная характеристика пяти типовых областей применения языков программирования, достаточно востребованных современным человеческим обществом. Подробно описываются формы и содержание критериев для оценки эффективности языков и рассматриваются способы построения этих критериев.

Вторая глава знакомит со средой, в которой выполняется компьютерная программа, и процессом перевода текста программы в исполняемую форму. Изложение начинается с изучения аппаратной организации компьютеров, где выделяется принцип программного управления и рассматривается реализующая его структура, а затем описывается порядок функционирования компьютера. Вводится понятие виртуальной машины, устраняющей разрыв между языком, который «понимает аппаратура», и языком программирования, на котором написана программа. Далее это понятие распространяется на иерархию виртуальных машин, упрощающую взаимодействие между аппаратным уровнем и программными уровнями компьютерной системы. После чего объясняются возможности и категории специальных программ-переводчиков (компиляторов и интерпретаторов), занимающихся трансляцией — переводом текстов программ в форму, воспринимаемую аппаратурой компьютера. Дается развернутая характеристика всех этапов трансляции: лексического анализа, синтаксического анализа, семантического анализа, оптимизации, генерации кода и редактирования связей.

Третья глава отведена обсуждению различных разновидностей языков программирования: императивных, функциональных, логических и объектно-ориентированных языков. По каждой разновидности приводится отдельное описание одного-двух языков. Кроме того, дается характеристика языка гипертекстовой разметки HTML.

Самое пристальное внимание обращено на скриптовые языки. Помимо их общей характеристики, включены справки по языкам Perl, JavaScript, PHP, Python, Ruby и Lua. Глава завершается рассмотрением таких гибридных языков разметки/программирования, как XSLT и JSP.

Предметом внимания четвертой главы являются разные формы записи выражений в языках программирования: префиксная, постфиксная, инфиксная и смешанная нотации. Обсуждается содержание и разновидности самой популярной и странной операции — присваивания. В завершение поясняется порядок вычисления операндов в выражении, всесторонне анализируется проблема побочных эффектов вычисления.

Пятая глава поясняет весь спектр операторов, применяемых в языках программирования: операторы для создания линейных, разветвляющихся и циклических программ. Рассматриваются простые операторы и операторы перехода, после чего вводится понятие потока управления в программе и все остальные средства анализируются с позиций обеспечения его структурированности. Для удобства изложения операторы разветвления разделены на условные операторы (ветвление на два направления) и операторы выбора (ветвление на много направлений), а операторы циклов классифицируются по основанию «число повторений». Среди циклов с заданным числом повторений отмечены циклы с повторением по данным, достаточно популярные в настоящее время. В свою очередь, среди операторов с не заданным числом повторений выделяются бесконечные циклы, циклы с предусловием и циклы с постусловием. Все операторы иллюстрируются многочисленными примерами из популярных императивных, скриптовых, объектно-ориентированных и функциональных языков. В качестве поучительного примера средств управления с недетерминированной семантикой описываются охраняемые структуры Дейкстры.

Шестая глава отведена основам и месту синтаксиса в языке программирования: характеризуются особенности определения языка, показатели качества его синтаксиса; подробно обсуждаются синтаксические элементы языка, после чего вводится понятие абстрактного синтаксиса и абстрактных синтаксических деревьев. Далее поясняется термин «контекстно-свободная грамматика» (как средство для точного и формального задания синтаксиса) и описываются самые популярные формы для представления грамматики: обычная и расширенная формы Бэкуса—Наура, а также синтаксические схемы. Исследуются основные концепции, закладываемые в основу создания новой грамматики. Демонстрируется применение двух инструментов для выполнения грамматического разбора: графических деревьев разбора и текстовых выводов. Приводятся наглядные примеры выявления синтаксических проблем: неоднозначности повисшего else, стороннего влияния таких элементов, как точки с запятой и пустые операторы.

Седьмая глава — это обзор формальных подходов к описанию семантики языков программирования. Вначале поясняются синтезируемые атрибуты и атрибутные грамматики, которые являются расширениями обычных, синтаксических грамматик на область семантики. Затем дается характеристика операционной семантики, максимально использующей воображение человека в роли виртуальной машины. Значительное место и внимание уделено аксиоматической семантике — наиболее развиваемому на практике подходу к автоматическому доказательству корректности

программы. Рассматривается принятая в нем схема доказательства, использование правил вывода, опирающихся на постусловия и предусловия; приводятся примеры подобных правил для наиболее популярных конструкций языка. В частности, описывается понятие инварианта цикла и исследуются методики его определения. В заключение объясняются основные идеи денотационного подхода к заданию семантики, где во главу угла поставлены рекурсивные функции отображения отдельных конструкций языка в область семантических значений. Демонстрируются примеры функций семантического отображения для чисел, выражений, операторов присваивания и циклов.

Восьмая глава раскрывает круг вопросов, связанных с понятиями объект и тип данных. В этой главе рассматриваются: объекты данных, переменные и константы как разновидности объектов, типы данных в качестве инструмента классификации и контроля объектов, объявления как описания объектов в программе. Поясняется концепция механизмов статического и динамического контроля типов. Последовательно обсуждаются шесть атрибутов переменной, в состав которых входят: имя, тип, адрес, значение, время жизни, область видимости. Исследуется статическое и динамическое связывание атрибутов переменной. Значительное внимание уделяется классификации переменных по месту размещения и времени жизни. Кроме того, здесь определяются и объясняются понятия «система типизации данных» и «тип выражения».

Девятая глава содержит материал по семейству элементарных типов данных под общим названием скалярные типы: перечислениям, целым и вещественным типам, десятичным числам. Вводится понятие поддиапазона, который ограничивает диапазон значений существующего базового типа, обсуждается логический тип и символьные типы. В завершение комментируется стиль программирования на языке C, основанный на неявном преобразовании типов.

Десятая глава освещает составные типы данных: массивы, строки символов, записи, множества, кортежи и списки. Массивы рассматриваются самым пристальным образом. В частности, излагаются следующие вопросы: разновидности массивов (с позиций размещения в памяти и времени жизни), их инициализация, атрибуты и операции простых массивов и их развитых (например, в скриптовых языках) версий. Многомерные массивы подразделяются на прямоугольные и массивы массивов, поясняется специфика доступа к сечениям массивов, а также доступ по содержанию к ассоциативным массивам. Помимо обычных записей описываются объединения и вариантные записи, анализируется их надежность.

Одиннадцатая глава представляет типизированные средства косвенного доступа к объектам данных — указатели. Здесь обсуждаются основные характеристики, понятия указателей, операции над указателями; поясняется специфика построения на их основе динамических связных структур; рассказывается о сокращении затрат времени при работе (посредством указателей) со сложными структурами. Исследуются такие проблемы их применения, как повисшие указатели и утечки памяти. Анализируются, сравниваются безопасные и гибкие аппараты указателей, реализованные в различных языках. Описывается особая разновидность указателя — ссылка, введенная в языке C++, поясняются преимущества использования ссылок. В заключение комментируются вопросы реализации указателей.

В двенадцатой главе разъясняются решения по поводу совместимости форматов для различных объектов данных. В ходе подобных решений рассматривают

три возможности: эквивалентность типов данных, явное приведение и неявное приведение форматов значений. Здесь поясняется структурная и именная форма эквивалентности, причем обсуждается как сильная (строгая), так и слабая (нестрогая) именная эквивалентность. Явные приведения форматов требуют участия программиста, а неявные приведения система выполняет самостоятельно. Далее разбираются достоинства и недостатки перечисленных разновидностей приведения. В качестве примера дополнительно описываются явные приведения в языке C++. Кроме того, оцениваются уровни типизации популярных языков программирования, свидетельствующие об их защищенности от ошибок типов.

Тринадцатая глава решает задачу презентации обширного комплекса вопросов, связанных с подпрограммами (процедурами и функциями): от их определения до применения. Поясняются все основные способы передачи параметров в подпрограммы (и из подпрограмм), а также сопутствующие вопросы (организация области видимости, проверка типов параметров и т. д.). Объявления и применения подпрограмм иллюстрируются многочисленными примерами на различных языках программирования. Достаточно подробно освещена проблема полиморфизма подпрограмм во всех его проявлениях: параметрический полиморфизм, полиморфизм подтипов, перегрузка и неявное приведение. Проанализирована реализация полиморфизма в различных языках программирования. Помимо обычных подпрограмм, рассмотрены различные реализации родовых подпрограмм, обеспечивающих параметрический полиморфизм. Детально описан особый режим использования подпрограммы — замыкание.

Четырнадцатая глава определяет базовые понятия и механизмы взаимодействия с подпрограммами, затрагивающие их размещение в памяти и связывания, которые происходят при выполнении подпрограмм. Рассказывается об эффекте вложения областей видимости, который возникает при помещении подпрограммы в контекст программы, затем внимание концентрируется на активации подпрограммы в ходе ее вызова. Поясняется поток управления между активациями подпрограмм, иллюстрируемый деревом активации, а также формат записи активации. Описываются особенности размещения записи активации в куче и стеке, а также освобождение кучи (стека) после завершения выполнения подпрограммы. Далее приводятся примеры управления подпрограммами в языках C и Pascal, а также объясняется методика оптимизации взаимодействия с подпрограммой на основе дисплеев.

Пятнадцатая глава вводит в круг понятий, связанных с крупноблочным строительством модулей программы. Вначале обсуждается специфика инкапсуляции — занесения данных и подпрограмм в единую капсулу, итогом которой становится абстрактный тип данных (АТД). Функциональные возможности АТД превышают возможности одиночной подпрограммы. Далее приводятся примеры создания и использования обычных АТД (в различных языках программирования). Область применения АТД увеличивается за счет параметрического полиморфизма. Таким способом разрабатываются родовые АТД, которые также поясняются примерами. Следующий шаг в этом же направлении приводит к более крупным контейнерам (физическим контейнерам), в каждый из которых можно положить несколько типов данных. После рассмотрения этого шага внимание читателя переключается на пространства имен: логические контейнеры, поддерживающие параллельную и независимую разработку очень больших программ.

Шестнадцатая глава ориентирована на понятия объектно-ориентированного подхода к программированию. Последовательно анализируются объекты и классы как основные строительные элементы объектно-ориентированных программ. Значительное внимание уделяется описанию отношений между объектами и классами. Затем обсуждаются возможности и природа наследования, его корректность, формы наследования. Описываются и исследуются самые разнообразные вопросы: синтаксис пересылки сообщений; статическое и динамическое связывание сообщения и метода; разновидности переопределения методов (замещение и уточнение). Далее рассматриваются особенности объектно-ориентированного программирования на различных языках. Дополнительно поясняются особенности и базовые понятия аспектно-ориентированного подхода, приводятся примеры аспектного программирования на языке AspectJ.

Семнадцатая глава обучает основным характеристикам и понятиям, связанным с исключениями; в частности, освещаются предопределенные и определяемые программистом исключения, обработчики (ловушки) исключений и возникновение исключений. Формулируются и обсуждаются этапы работы с определяемыми исключениями, потоки управления при обработке исключений, организация многоуровневой системы исключений. Приводятся и анализируются средства обработки исключений в четырех языках программирования: Ada, C++, Java и C#.

Восемнадцатая глава отражает проблематику файлов, потоков и организации ввода-вывода. После пояснения задач аппарата ввода-вывода и классификации файлов здесь описываются средства как процедурного, так и объектно-ориентированного (потокового) ввода-вывода.

Девятнадцатая глава начинается с описания основных понятий параллельного программирования — процессов, потоков и их взаимодействия. После чего в терминах языка Ада рассматриваются решения следующих классических проблем: синхронизация процессов с помощью разделяемых переменных (на основе семафоров, мониторов, защищенных объектов); синхронизация процессов с помощью сообщений; «развязка» взаимодействия процессов при рандеву. Затем изучаются потоки в языках Java и C#.

В двадцатой главе излагаются вопросы, связанные с функциональным подходом к написанию программ. Анализируются специфические особенности функциональных языков программирования. На примере языка Scheme рассматриваются классические понятия и средства функционального программирования: выражения, специальные формы, создание функций, динамическая проверка типа, хвостовая и не хвостовая рекурсия, структуры данных, специфика приемов программирования, функции высшего порядка, статическая область видимости, настройка — специализация языка. Далее на примере языка ML освещается функциональный подход со статической типизацией, подчеркиваются концептуальные вопросы, которые приходится при этом решать. Здесь же поясняется понятие карризации функций, приводятся примеры создания карризованных функций. В заключение описывается проблематика отложенных вычислений: нестрогие функции, специальные формы для отложенных вычислений, передача по необходимости, мемоизация, ленивые вычисления.

Цель двадцать первой главы — ввести основные понятия логического программирования и познакомить с базовыми средствами языка Prolog. Вначале рассма-

триваются математические основы логического программирования: исчисление предикатов. Затем обсуждаются трудности, возникающие при использовании исчисления предикатов в автоматизированных дедуктивных системах, и показываются пути их решения на основе формул Хорна и механизма «резолюции–унификации». Приводится общий обзор языка логического программирования Prolog, поясняются его нотация и структуры данных, процесс выполнения программы, арифметика, унификация и стратегия поиска, а также циклы и структуры управления. В завершение описаны проблемы применения языка Prolog в логическом программировании: проверка вхождения при унификации, замкнутость мира, своеобразие логического отрицания, ограниченность формул Хорна при описании логики утверждений, принудительное внесение информации управления.

Учебник предназначен для студентов инженерного, бакалаврского и магистерского уровней компьютерных специальностей, может быть полезен преподавателям и исследователям/разработчикам трансляторов и другого программного обеспечения.

Вот и все. Насколько удалась эта работа — судить вам, уважаемый читатель.

Благодарности

Прежде всего, мои слова искренней любви родителям — Нине Николаевне и Александру Ивановичу Орловым (светлая им память).

Самые теплые слова благодарности моей семье, родным, любимым и близким мне людям — Лизе, Иванне, Жене. Без их долготерпения, внимания, поддержки, доброжелательности и сердечной заботы эта книга никогда бы не была написана.

Выход в свет этой работы был бы невозможен вне творческой атмосферы, бесчисленных вопросов и положительной обратной связи, которую создавали мои многочисленные студенты.

Особую благодарность хочется выразить руководителю проекта Андрею Юрченко, благодаря профессионализму, добросовестности, интеллигентности и настойчивости которого и состоялось данное издание.

Автор искренне признателен талантливым сотрудникам издательства «Питер».

И конечно, огромное спасибо моим коллегам, всем людям, которые принимали участие в путешествии по городам, улицам и бесконечным переулкам страны ЯЗЫКИ ПРОГРАММИРОВАНИЯ.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1

Определение и проблемы языков программирования

В этой главе начинается знакомство с языками программирования. Во-первых, обсуждаются причины, по которым следует изучать общие понятия, связанные с использованием и созданием языков программирования. Во-вторых, поясняется предлагаемый подход к изучению (принцип абстракции-конкретизации). Далее вводятся определения языка с трех точек зрения: семиотической, практической и технологической. На наш взгляд, они способствуют формированию необходимой панорамы представления о таком сложном объекте изучения, как язык программирования. Затем дается обзорная характеристика пяти типовых областей применения языков программирования, достаточно востребованных современным обществом. Кроме того, в главе описываются формы и содержание критериев для оценки эффективности языков и рассматриваются способы построения этих критериев.

Для чего нужно изучать принципы построения языков программирования

Цель нашего учебника — обеспечить систематическое изучение теории и практики современных языков программирования (ЯП): принципов их построения, оценки и использования [27].

Зачем могут понадобиться знания о языках программирования?

1. Понимание общих идей построения и применения языка — это ключ к легкому и глубокому освоению конкретного языка как основного профессионального инструмента программиста.
2. Понимание принципов построения языков — необходимое условие предоставления пользователю максимума услуг при минимуме затрат.
3. Изучение ЯП дает знание и понимание разнообразных подходов к программированию. Каждый ЯП — это своя философия, свой взгляд на деятельность программиста, отражение определенной технологии программирования.

   Приведем несколько примеров:

   - **Algol** (международная группа ученых, руководитель Петер Наур) — во главу угла был поставлен академический подход. Творцы языка акцентировали

внимание на строгости определения и изяществе языковых конструкций. Самое важное, считали они, — это ясно и точно сформулировать алгоритм. В стороне остались технические вопросы: ввод-вывод, компоновка программ из модулей.

- **Fortran** (фирма *IBM*, руководитель Джон Бэкус) — яркий пример прагматического подхода. Строгости и изяществу языка здесь уделялось мало внимания. Основной акцент был сделан на модульности, организации ввода-вывода.
- **Basic** (Дартмутский колледж, Джон Кемени и Томас Курц) — реализовал диалоговый подход к взаимодействию с компьютером. Главными целями были объявлены простота языка, дружественность к пользователю. Мало того, язык поддерживал совсем иной принцип взаимодействия пользователя с компьютером. Предлагалось обеспечить каждого пользователя (неслыханное дело!) своим терминалом для взаимодействия с машиной.

4. Понятия и тенденции в области ЯП точно отражают особенности программирования как науки, искусства и ремесла. Фактически мы обсуждаем основные принципы и понятия программирования с языковой точки зрения.
5. Эти знания позволят оценить и выбрать ЯП для выполнения конкретного проекта, учесть затраты на его освоение и распространение написанных программ.

Аппарат абстракции-конкретизации

Существует огромное количество языков программирования, среди которых выделяют дюжину самых популярных. Каждый язык демонстрирует свою специфику, имеет свои достоинства и недостатки. Очень легко потеряться в море информации о реальных языках программирования. Чтобы этого не случилось, будем изучать в основном модели языков, активно использовать моделирование как аппарат абстракции-конкретизации [13, 29].

Создавая понятие (в интересах какой-либо задачи), мы отвлекаемся (абстрагируемся) от несущественных свойств реальных объектов, фиксируя только существенные свойства.

Пример. Рассмотрим модель часов. Выделим только одно свойство — «показывать время». В этой модели мы, к примеру, отказываемся от остальных свойств реальных часов: формы, цвета, материала, изготовителя.

Приводя пример, мы, напротив, конкретизируем абстрактное понятие, снабжая его второстепенными, но важными в данной ситуации деталями:

- Конкретное выполнение процедуры происходит при конкретных значениях ее параметров.
- Конкретный пример языка программирования — C++ — включает в себя конкретные конструкции.

Абстрактное понятие (модель) проще конкретного объекта, оно отражает лишь некоторую точку зрения на объект.

При решении другой задачи точка зрения на объект может быть другой, может потребоваться другая модель того же объекта.

Приведем ряд примеров:

- Модель часов в качестве украшения включает свойства: форму, цвет, фирму, размер. Отказываемся от свойства «показывать время».
- Модель размещения процедуры в памяти. Отказываемся от смысла выполняемых в ней действий.

ВНИМАНИЕ

Непроизвольный переход от одной модели к другой может быть источником недоразумений!

Вы говорите, что часы «плохие», потому что некрасивые, а я говорю «хорошие», так как они отлично работают. Если мы решаем «часы хорошие или плохие?», нам надо задаться критерием, уместным в рамках конкретной модели.

Итак, аппарат абстракции-конкретизации — это инструмент, которым надо пользоваться умело и осторожно. Мы будем подробно обсуждать источники недоразумений, возникающих при применении абстракции-конкретизации для изучения языков программирования. Однако предварительно сформулируем панораму определений предмета изучения — языка программирования.

Исходное определение языка программирования

Язык программирования (ЯП) — это знаковая система для планирования поведения компьютера.

Знаковая система — это совокупность соглашений, определяющих набор знаковых ситуаций. *Знаковая ситуация* — первичное понятие семиотики, ее элементами являются знак и денотат (рис. 1.1).

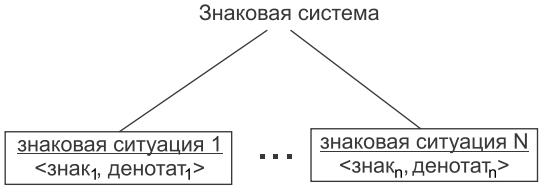


Рис. 1.1. Структура знаковой системы

Знак — это обозначение или имя, *денотат* — значение, смысл.

Примеры знаков и денотатов приведены в табл. 1.1.

Правила, определяющие структуру допустимых знаков, называют *синтаксисом* языка [17, 46–48, 50, 82, 88].

Правила, определяющие денотаты, которые соответствуют допустимым знакам, называют *семантикой* языка [17, 33, 46, 48, 50, 67, 69, 82, 88].

Опираясь на введенные понятия, можно предложить модифицированную формулировку языка программирования.

Таблица 1.1. Знаки и денотаты

Знак	Денотат
Сообщение	Смысл
Буква	Звук
Слово	Понятие
Sqrt	Вычисление квадратного корня
If-then-else	Разветвление вычислений

Язык программирования — это правила образования знаков (синтаксис) и согласованные с ними правила образования денотатов (семантика).

В ЯП знаки — это элементы программ, а денотаты — характеристики поведения исполнителя (компьютера).

Практическое определение языка программирования

Язык программирования — это средство общения между человеком (пользователем) и компьютером (исполнителем).

С помощью ЯП формируются сообщения для компьютера. Эти сообщения должны быть понятны компьютеру. Возможные ошибки разделяют на синтаксические, семантические и прагматические (рис. 1.2).

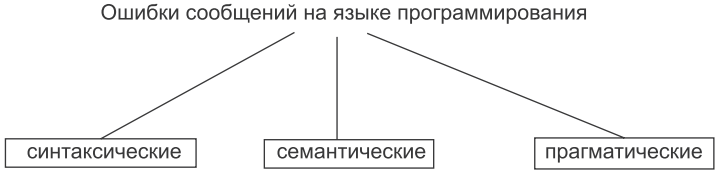


Рис. 1.2. Классификация ошибок сообщений

Синтаксические ошибки возникают, если автор сообщения подразумевает одну структуру сообщения, а получатель — другую. Устраняются легко. В ЯП таких ошибок не бывает, так как их создатели используют *принцип синтаксической однозначности* (правильный текст должен иметь единственную допустимую структуру).

Семантические ошибки связаны с искажением смысла элементов сообщения, а *прагматические ошибки* — с искажением цели (назначения) элемента или сообщения.

Прагматические ошибки — самые трудные. Их примеры: часы хорошие-плохие, выяснение смысла программы без знания ее назначения.

Для исключения ошибок необходимо, чтобы отправитель и адресат пользовались:

- одинаковыми правилами разложения сообщения на составляющие (предмет синтаксиса);
- согласованными правилами, придающими сообщению смысл (предмет семантики);
- согласованными целевыми установками (предмет прагматики).

Технологическое определение языка программирования

Язык программирования — инструмент для производства программных услуг.

Программирование нацелено на обслуживание пользователя. Принцип хорошего обслуживания: «Клиент всегда прав». Это означает, что программы должны быть дружественными к пользователю.

Программы должны быть:

- надежными (содержать мало ошибок);
- устойчивыми (сохранять работоспособность в неблагоприятных условиях эксплуатации, ограниченности ресурсов, перегрузок, ошибок пользователя, сбоев и отказов аппаратуры);
- заботливыми (должны уметь объяснять свои действия и ошибки пользователя).

Известно, что создание программ — дорогое занятие. С современной точки зрения, аппаратура компьютера по относительной стоимости сравнима с упаковкой, в которую «заворачивается» программное изделие. Почему? Причина — сложность программирования. Известны два источника сложности [3, 13].

Семантический разрыв — разрыв между уровнем элементарных операций компьютера и уровнем потенциально возможных услуг. Компьютер — очень быстрый исполнитель. Отсюда соблазн программировать его на оказание сложных услуг (с помощью очень сложных программ). В то же время возможности человека при работе со связанными объектами ограничены числом ИНГВЕ = 7 ± 2. Выход из положения — создавать для определенного класса услуг подходящий ЯП (со сложными операциями) и программировать на таком ЯП.

Незнание компьютером реального мира. Поэтому компьютер не может контролировать соответствие указаний программиста поставленным задачам (целям), ради достижения которых они сформированы.

Примеры серьезных программных ошибок:

- Космический аппарат «Маринер-1» (22 июля 1962 г.). Цель — Венера. Уничтожен через 294,5 секунды после старта.
- Космический аппарат «Маринер-2» (27 августа 1962 г). Источник ошибки — пробел и пропуск запятой в Фортран-программе управления (оператор цикла DO).
- Ракета-носитель «Ариан-5» (4 июня 1996 г.). Взорвана при старте. Стоимость ошибки — $500 000 000.
- Конвейерные отключения электричества на северо-востоке Северной Америки (14 августа 2003 г.). Стоимость — 10 миллиардов долларов.

Средство борьбы с семантическим разрывом — аппарат абстракции-конкретизации языка. Этот аппарат — основа ориентации на проблему. Например, в языке Fortran средство абстракции — подпрограмма, а соответствующее средство конкретизации — обращение к ней с фактическими параметрами.

Средство борьбы с незнанием реального мира — аппарат прогноза-контроля языка. Этот аппарат — основа повышения надежности и устойчивости программ. Подобный аппарат в языке ассемблера очень слаб. В языке Fortran средством про-

гнозирования являются встроенные типы данных, соответствующий контроль предусмотрен семантикой языка, но средств управления таким контролем нет. В языках искусственного интеллекта такой аппарат прямо предназначен для предоставления знаний о мире, целях деятельности и контроля действий (как программы, так и пользователя).

Теперь мы можем сформулировать *технологический критерий качества ЯП*: язык тем лучше, чем проще производство (на его основе) программных услуг.

Области применения языков программирования

Современное человеческое общество все больше и больше зависит от компьютерных систем, управляемых программами. Эти системы весьма настойчиво проникают во все сферы человеческой жизни. Они буквально окружают человека и на работе, и дома, и на отдыхе [47, 48, 82, 88].

Известно, что аппаратура современных компьютерных систем очень сложна, и вместе с тем считают: сложность программного обеспечения превосходит аппаратную сложность более чем на порядок. Ф. Брукс, известнейший авторитет в данной области, утверждает: «Сложность программного обеспечения является существенным, а не второстепенным свойством» [5].

С одной стороны, программы абстрактны и нематериальны. Они не имеют физической природы и присущих этой природе ограничений, кажутся человеку-творцу очень податливой «глиной», из которой можно «вылепить» все, что угодно. С другой стороны, сложность программного обеспечения может превосходить возможности человеческого разума.

Основной инструмент для создания программ — язык программирования. Выбор подходящего языка программирования для решения задачи определяется множеством причин, но главной причиной является принадлежность задачи к конкретной предметной области. Обсудим наиболее представительные области, в которых применяются языки программирования.

Научные вычисления

Первые компьютеры, появившиеся в середине XX века, изначально создавались для научных целей. Задачи из этой области сводились к решению различных математических уравнений. К ним относились задачи численного анализа, решения дифференциальных и интегральных уравнений и задачи статистики. Обычно научные программы обрабатывают простые структуры данных и производят большое количество арифметических вычислений, выполняемых над числами с плавающей точкой. Наиболее часто употребляются такие структуры, как массивы и матрицы; из управляющих структур чаще других используются циклы со счетчиком и условные операторы. Языки «научного» программирования разрабатывались именно для удовлетворения таких потребностей. Здесь всегда доминировал Fortran, поскольку его синтаксис был очень близок к математическому языку ученых. Язык Algol 60

и большинство его потомков также предназначались для подобных целей, хотя могли применяться и в других родственных областях. Однако для научных вычислений особо важна эффективность, а ни один из созданных языков не оказался лучше языка Fortran. В XXI веке Fortran по-прежнему не сдает своих позиций, однако языки Java и C++ достаточно успешно конкурируют с его современной версией — Fortran 2008.

Обработка деловой информации

Задачи этой области, называемые бизнес-задачами, ориентированы на учет поступления заказов, управление ресурсами и персоналом, деловое планирование, анализ риска и оценку возможных вариантов, а также начисление зарплаты. Бизнес-программы обеспечивали считывание больших объемов данных, собранных за длительный период времени, и сохраняли их на магнитных лентах и дисках, а также формировали новые данные, обновляемые в результате небольших преобразований. Языки бизнес-программирования нацеливались на генерацию отчетов со сложной структурой, точные способы описания и хранения десятичных чисел и символьных данных, а также на арифметические действия с десятичными числами. Для этих задач был разработан язык Cobol, который и сейчас широко используется. Разработчики этого языка приложили немало усилий для реализации корректной обработки данных.

В наши дни основным языком бизнес-программирования по-прежнему считается Cobol, хотя иногда применяют и другие языки (например, C, C++). Однако позицию языков при обработке деловой информации сильно потеснили электронные таблицы, придуманные для персональных компьютеров. Если раньше на создание обычной программы делового планирования уходило несколько месяцев, то теперь аналитик может за несколько часов *сформировать* много электронных таблиц, решающих аналогичные задачи.

Дополнительным инструментом деловой обработки стали языки четвертого поколения 4GL (Fourth Generation Languages). Языки 4GL — это непроцедурные языки, специализированные под конкретные задачи обработки деловой информации и рассматривающие программу как спецификацию требуемого результата, а не как спецификацию действий, приводящих к достижению этого результата. Поддерживающая язык 4GL среда переводит спецификацию результата в машинно-выполнимую программу. Языки 4GL обеспечивают быстрое создание оконного интерфейса и простой доступ к содержимому базы данных, генерацию графических входных форм-бланков и выходных отчетов.

Искусственный интеллект

Искусственный интеллект (ИИ) — это предметная область, главная цель которой состояла в моделировании мыслительной деятельности человека. Отличительной особенностью возникавших здесь задач являлись символьные, а не числовые вычисления. Под символьными вычислениями подразумевается обработка не чисел, а символов. Оказалось, что символьные вычисления удобнее производить с помощью связных списков данных, а не массивов. Достаточно часто символьные

вычисления требуют большей гибкости, чем другие предметные области. Например, в некоторых задачах целесообразно создание целых фрагментов программы в период ее выполнения.

Основным инструментом производимых вычислений здесь считается функция, а используемые языки программирования называются функциональными языками. Родоначальником языков функционального программирования стал язык LISP, придуманный ученым Д. Маккарти (1965). Последователями языка LISP являются языки Scheme, Haskell и т. д. В 70-х годах для решения задач ИИ начали применять и альтернативный подход на основе языка логического программирования Prolog.

Системная область

Все задачи системной области связаны с созданием и развитием операционных систем для компьютеров — системным программным обеспечением (ПО). Понятно, что системное ПО широко используется, имеет доступ ко всем функциональным возможностям и ресурсам аппаратуры и должно эффективно функционировать. Как следствие, применяемые языки программирования должны обеспечивать быстрое выполнение программ и взаимодействие с аппаратными средствами.

Вначале для достижения максимальной эффективности системное программирование велось на языке ассемблера.

В настоящее время повсеместно применяется язык C и его более новый вариант C++. Например, операционная система UNIX полностью написана на языке C. Язык C обеспечивает очень эффективное выполнение программ и позволяет программисту получить полный доступ как к операционной системе, так и к аппаратуре. Он не обременяет программиста большим количеством ограничений. Профессионалы, занимающиеся системным программированием, обычно думают, что подобные ограничения им не нужны. Тем не менее некоторые специалисты считают язык C слишком опасным для программирования больших и важных систем.

Еще один язык — Ada — тоже проектировался для применения в системной области, однако из-за ряда субъективных и объективных причин не получил здесь статуса основного языка. Он востребован в системах аэрокосмического и военного назначения.

Близкой предметной областью считается управление технологическими процессами и оборудованием. С появлением дешевых микропроцессоров, используемых в автомобилях, светофорах, стиральных машинах, видеоиграх и электронных часах, возросла необходимость в языках, позволяющих создавать программы для работы в реальном времени. К таким языкам относятся C, Ada и C++.

Веб-обработка

Появление Всемирной паутины (WWW — World Wide Web) оказало сильное воздействие на географию компьютерных взаимодействий. По всему миру создаются серверы информационных архивов. Для получения информации пользователи подключаются к этим серверам через Интернет, а для ее обработки используют локальные клиентские машины. Чтобы пользователь мог загрузить программу,

а поставщик программного обеспечения имел возможность получить плату за ее использование, стал необходим язык программирования, позволяющий вести диалог между клиентским компьютером и сервером. Развитие электронной коммерции находится в полной зависимости от языков с такими возможностями.

Изначально веб-страницы были статическими документами: можно было лишь просмотреть текст или рисунки. Создавались страницы с помощью языка разметки HTML, который не является языком программирования. Для доступа к другой странице пользователь мог щелкнуть на ее адресе URL (Uniform Resource Locator). Однако для развития электронной коммерции информация должна передаваться в обоих направлениях между клиентской машиной и сервером, поэтому веб-страницы должны были стать активными, динамическими. Подобные возможности обеспечиваются такими языками программирования, как Perl, PHP и Java.

Использование WWW снова поставило перед языками проблему производительности:

1) линии связи пользователя с Интернетом, как правило, ограничивают скорость передачи данных;
2) из-за подключения большого количества пользователей сервер может оказаться перегружен.

Для исключения влияния таких ситуаций можно обрабатывать информацию на клиентской машине, а не на сервере. Чтобы разгрузить сервер за счет клиентской машины, он должен переслать клиенту небольшую исполняемую программу. Но ведь серверу неизвестно, каким компьютером является клиентская машина, поэтому не ясно, какого вида должна быть исполняемая программа. В главе 2 мы обсудим то решение, которое предлагает в этой ситуации язык Java, созданный для работы в среде WWW. Кроме того, заметим, что исполняемые программы (аплеты) языка Java были специально разработаны для обеспечения конфиденциальности частной жизни пользователя. Аплету, например, не разрешается выполнять запись или удаление файлов на клиентской машине.

Критерии эффективности языков программирования

Язык программирования можно определить множеством показателей, характеризующих отдельные его свойства. Возникает задача введения меры для оценки степени приспособленности ЯП к выполнению возложенных на него функций — меры эффективности.

Эффективность определяет степень соответствия ЯП своему назначению. Она измеряется либо количеством затрат, необходимых для получения определенного результата, либо результатом, полученным при определенных затратах. Произвести сравнительный анализ эффективности нескольких ЯП, принять решение на использование конкретного языка позволяет критерий эффективности.

Критерий эффективности — это правило, служащее для сравнительной оценки качества различных ЯП. Критерий эффективности можно назвать правилом предпочтения сравниваемых вариантов [28].

Строятся критерии эффективности на основе частных показателей эффективности (показателей качества) Способ связи между частными показателями определяет вид критерия эффективности.

В качестве частных показателей обычно выступают:

- читабельность (легкость чтения и понимания программ);
- легкость создания программ (удобство языка для создания программ в выбранной области);
- надежность (обеспечение минимума ошибок при работе программ);
- стоимость (суммарная стоимость всего жизненного цикла языка);
- переносимость программ (легкость переноса программ из одной операционной среды в другую);
- универсальность (применимость к широкому кругу задач);
- четкость (полнота и точность официального описания языка).

В свою очередь, каждый из частных показателей зависит от множества разнообразных характеристик.

Рассмотрим самые важные из показателей более подробно [81, 84–86].

Читабельность

Одним из важнейших показателей качества языка программирования является легкость чтения и понимания программ, написанных на нем. По современным представлениям самый длинный период времени из жизненного цикла программ приходится на сопровождение, в ходе которого программы часто модифицируются. Поскольку читабельность программ определяет легкость сопровождения, ее считают существенной характеристикой качества программ и языков программирования.

Читабельность (Readability) языка программирования должна способствовать легкому выделению основных понятий каждой части программы без обращения к документации.

Простота. Простота сильно влияет на читабельность языка программирования. Язык должен предоставить простой набор конструкций, которые могут быть использованы в качестве базисных элементов при создании программы. Желательно обеспечить минимальное количество различных понятий с простыми правилами их комбинирования. Этому мешает наличие в языке нескольких способов описания одного и того же действия. Например, в языке C добавление единицы к целому числу можно записать четырьмя способами. Сильное воздействие на простоту оказывает синтаксис языка: он должен прозрачно отражать семантику конструкций, исключать двусмысленность толкования. Предельно лаконичный синтаксис удобен при написании программы, однако усложняет ее модификацию, поскольку в программе нелегко разобраться. Здесь нужен разумный компромисс — простота не должна быть чрезмерной, не должна приводить к загадкам расшифровки.

Ортогональность. Ортогональность означает, что любые возможные комбинации различных языковых конструкций будут осмысленными, без непредвиденных огра-

ничений или неожиданного поведения, возникающих в результате взаимодействия конструкций или контекста использования.

Например, предположим, что ЯП содержит три элементарных типа данных (целый, вещественный с плавающей точкой и символьный), а также две конструкции данных (массив и указатель). Если обе конструкции могут применяться к этим типам и самим себе, то говорят об их ортогональности, обеспечивающей создание большого количества структур данных.

Когда конструкции языка ортогональны, язык легче выучить и использовать для чтения и создания программ, ведь в нем меньше исключений и специальных случаев, требующих запоминания.

Приведем примеры недостатка ортогональности в языках:

- В языке Pascal функции могут возвращать значения только скалярного или указательного типов, а в языках C и C++ — значения всех типов, за исключением массивов (трактовка массивов в этих языках отличается от трактовки остальных типов). В функциональных языках, языках Ada и Python этот недостаток ортогональности устранен.
- В языке C локальные переменные могут быть определены только в начале блока (составного оператора), а в C++ переменные определяются в любом месте блока (но, конечно, перед использованием).
- В языке Java величины скалярных типов (символьного, целого, вещественного и т. д.) не являются объектами, а величины всех остальных типов считаются объектами. Скалярные типы называют примитивными типами, а типы объектов — ссылочными типами. Примитивные типы используют *семантику значения* (значение копируется во время присваивания). Ссылочные типы используют *семантику ссылки* (присваивание формирует две ссылки на один и тот же объект). Кроме того, в языке Java коллекции объектов и коллекции примитивных типов трактуются по-разному. Среди коллекций в Java только массивы могут содержать примитивные значения. Примитивные значения могут вставляться в коллекции других типов лишь в капсулах объектов-оболочек. В языках Smalltalk и Python, напротив, все величины являются объектами, а все типы — ссылочными типами. Таким образом, в этих языках применяется лишь семантика ссылки и все коллекции объектов создаются в ортогональном стиле.

Излишняя ортогональность может стать источником проблем. Например, из-за разрешения полной ортогональности в ходе компиляции программы не генерируются ошибки, даже при наличии комбинаций, которые логически не согласованы или крайне неэффективны при выполнении.

Следовательно, простота языка должна быть результатом комбинирования небольшого числа элементарных конструкций и ограниченного применения понятия ортогональности.

Структурированность потока управления в программе. Порядок передач управления между операторами программы должен быть удобен для чтения и понимания человеком. Речь идет об ограниченном использовании оператора безусловного перехода goto и применении специальных структур управления. Более детально эта тема раскрывается в главе 5.

Легкость создания программ

Легкость создания программ (Writability) отражает удобство языка для написания программ в конкретной предметной области. Поскольку в каждой предметной области программы имеют свою специфику, очень важно выбирать язык, который ее учитывает. Например, если речь идет об исследовании искусственного интеллекта, следует использовать язык Prolog или LISP, а при решении задач научных вычислений — язык Fortran.

Как и следовало ожидать, характеристики, от которых зависит этот показатель, во многом пересекаются с характеристиками читабельности, ведь чтение текста является неотъемлемым атрибутом как процесса создания, так и процесса понимания программы.

Концептуальная целостность языка. Концептуальная целостность языка включает в себя три взаимосвязанных аспекта: простоту, ортогональность и единообразие понятий. Простота языка предполагает использование минимального числа понятий. Ортогональность позволяет комбинировать любые языковые конструкции по определенным правилам. Единообразие понятий требует согласованного, единого подхода к описанию и использованию всех понятий.

Простота достигается не за счет отказа от сложных языковых конструкций, а путем устранения случайных ограничений на их использование. Так, при реализации массивов следует разрешить их объявление для любого типа данных, допускаемого языком. Если же, напротив, запретить массивы некоторого типа, то язык окажется скорее сложным, чем простым, поскольку основные правила языка изучить и запомнить проще, чем связанные с ними ограничения.

Простота уменьшает затраты на обучение программистов и вероятность ошибок, возникающих из-за неправильной интерпретации программистом языковых конструкций. Естественно, упрощать язык целесообразно лишь до определенного предела.

Если язык содержит большое количество разнообразных конструкций, то программисты могут просто не знать каждую из них. Это приводит к неправильному использованию одних возможностей и игнорированию других. В итоге программисты пишут не самые изящные и эффективные программы. Отсюда вывод: использовать небольшой набор элементарных конструкций и согласованных между собой правил их комбинирования намного удобнее, чем применять большое количество конструкций.

Еще раз напомним: ортогональность приемлема в разумных пределах. Чрезмерная степень ортогональности усложняет применение языка. Если разрешена любая комбинация элементарных конструкций, то логические ошибки программы могут просто остаться незамеченными, их не сможет выявить компилятор.

Естественность для приложений. Синтаксис языка должен способствовать легкому и прозрачному отображению в программах алгоритмических структур предметной области. Любой из типовых алгоритмов (последовательный, разветвляющийся, циклический, параллельный) имеет естественную структуру, которая должна поддерживаться программными операторами реализующего языка. Язык должен предоставлять структуры данных, операции и структуры управления, адекватные решаемой задаче. Естественность — это одна из важнейших причин

популярности того или иного языка. Язык, ориентированный на характерные черты предметной области, может сильно упростить создание для нее программных приложений. Приведем примеры языков с очевидной направленностью на решение конкретных классов задач: Prolog (поддерживает дедуктивные рассуждения), Perl (предназначен для записи различных сценариев).

Поддержка абстракций. Абстракция является инструментом определения сложных структур данных и действий, при использовании которого гарантируется простота, а также игнорируются многие второстепенные детали. Абстракция устраняет пробел между структурами данных и операциями, характерными для решения задачи, и конкретными структурами данных и операциями, встроенными в язык программирования. Например, при написании информационной системы института требуются абстрактные структуры данных *студент*, *курс*, *профессор*, *расписание* и абстрактные операции *записать студента на курс* и *спланировать расписание для курса*. Программист должен создать программные реализации этих абстракций с использованием реального языка программирования, в котором изначально они отсутствуют. После этого программные абстракции можно использовать как новые элементы и в других частях программы, не вникая в их фактическую реализацию. Ясно, что язык должен обеспечивать такую возможность. Например, в языке C++ поддержка абстракций существенно выше, чем в языке C.

Выразительность. Выразительность языка может характеризовать две возможности. С одной стороны, она означает наличие очень мощных средств для представления структур данных и действий, описывающих большой объем вычислений с помощью очень маленькой программы (языки APL, Snobol, Icon, SETL). С другой стороны, выразительность позволяет записывать вычисления в более удобной и компактной форме. Например, в языке C запись `x++` удобнее и короче записи `x = x + 1`. Аналогично, булевы операции `and then` и `or else` в языке Ada позволяют указать сокращенное вычисление булевых выражений. В языке Pascal циклы с известным количеством повторений проще создавать с помощью оператора `for`, чем с помощью оператора `while`. Несомненно, что все эти возможности облегчают разработку программ.

Надежность

В общем случае, надежность — это способность программы выполнять требуемые функции при заданных условиях и в течение определенного периода времени. Обычно уровень надежности характеризуется степенью автоматического обнаружения ошибок, которую обеспечивают транслятор и операционная среда выполнения программы. Надежный язык позволяет выявлять большинство ошибок во время трансляции программы, а не во время ее выполнения, поскольку это минимизирует стоимость ошибок.

Опишем факторы, имеющие сильное влияние на надежность программ.

Проверка типов. Принципиальным средством достижения высокой надежности языка является система типизации данных. В ходе проверки типов анализируется совместимость типов в программе. Разные языки обеспечивают разную полноту проверки типов. Достаточно слабой считают проверку типов в языке C. Языки

с динамической типизацией вообще относят эту проверку только к периоду выполнения программы. Наиболее полную проверку гарантирует языка Ada: в процессе компиляции программы проверяются типы практически всех переменных и выражений. Такой подход фактически устраняет ошибки типов при выполнении программы. В языках Pascal, Ada и Java диапазон изменения индексов является частью объявления массива и тоже подвергается проверке. Такая проверка очень важна для обеспечения надежности программы, поскольку индексы, выходящие за пределы допустимого диапазона, часто создают серьезные проблемы.

Обработка исключений. Исключением называют аварийное событие, которое обнаруживается во время выполнения программы (аппаратом исключений). В результате авария устраняется и программа продолжает работу. Подобный механизм значительно повышает надежность вычислений. Языки Ada, C++, C# и Java позволяют обрабатывать исключения, хотя во многих других языках этот механизм отсутствует.

Совмещение имен. Совмещением имен называют наличие нескольких разных имен у одной и той же ячейки памяти. Во многих языках переменным разрешается иметь по паре имен: обычное прямое имя и косвенное имя (на базе указателя). С одной стороны, совмещение имен может приводить к понижению надежности программы. С другой стороны, эта возможность повышает гибкость программирования и компенсирует недочеты жестких схем типизации.

В завершение отметим, что читабельность и легкость создания прямо влияют на надежность программы. Чем выше значения этих показателей, тем надежнее будет программа.

Стоимость

Суммарная стоимость языка программирования складывается из нескольких составляющих.

Стоимость выполнения программы. Она во многом зависит от структуры языка. Язык, требующий многочисленных проверок типов во время выполнения программы, будет препятствовать быстрой работе программы. Взгляды на этот фактор стоимости претерпели существенную эволюцию. В середине прошлого века фактор был решающим в силу высокой стоимости аппаратных средств компьютера и их низкой производительности. Большое значение придавалось применению оптимизирующих компиляторов, эффективному распределению регистров и механизмам эффективного выполнения программ. Сейчас считают, что стоимость (и скорость) выполнения программы существенна лишь для программного обеспечения систем реального времени. Программы реального времени должны обеспечивать быстрые вычисления управляющих воздействий на разнообразные управляемые объекты. Поскольку необходимо гарантировать определенное время реакции, следует избегать языковых конструкций, ведущих к непредсказуемым издержкам времени выполнения программы (например, при сборке мусора в схеме динамического распределения памяти). Для обычных приложений все снижающаяся стоимость аппаратуры и все возрастающая стоимость разработки программ позволяют считать, что скорость выполнения программ уже не столь критична.

Стоимость трансляции программы. Размер этой стоимости зависит от возможностей используемого компилятора. Чем совершеннее методы оптимизации, тем дороже стоит трансляция. В итоге создается эффективный код: резко сокращается размер программы и/или возрастает скорость ее работы.

Стоимость создания, тестирования и использования программы. Этот фактор стоимости удобно проиллюстрировать на примере среды для языка Smalltalk. Данная среда состоит из окон, меню, механизма ввода данных с помощью мыши и набора средств, позволяющих свободно оперировать со Smalltalk-программой. Здесь программное решение может быть спроектировано, закодировано, протестировано, изменено и использовано с минимальными затратами времени и сил программиста. По современным представлениям, наличие в языке развитых конструкций и структур является лишь одним аргументом, влияющим на широту его использования. Наличие же подходящей среды программирования существенно усиливает применимость слабого языка. Прежде всего, в среде программирования должна присутствовать надежная, эффективная и хорошо документированная реализация языка программирования. Специализированные текстовые редакторы, средства моделирования и управления конфигурацией, а также утилиты тестирования, отражающие особенности как самого языка, так и порядка его использования, — это мощные ускорители всех этапов разработки программ. В итоге минимизируются время и затраты, требуемые программисту на решение какой-либо задачи.

Стоимость сопровождения программы. Многочисленные исследования показали, что значительную часть стоимости используемой программы составляет не стоимость разработки, а стоимость сопровождения программы. Сопровождение — это процесс изменения программы после ее поставки заказчику. Сопровождение включает в себя:

- исправление ошибок (17% времени и стоимости);
- изменения, связанные с обновлением операционного окружения (18% времени и стоимости);
- усовершенствование и расширение функций программы (65% времени и стоимости).

В [87] утверждается, что соотношение между стоимостью сопровождения и стоимостью начальной разработки может быть разным в зависимости от предметной области, где эксплуатируется программа. Для прикладных программ, работающих в деловой сфере, стоимость затрат на сопровождение в основном сравнима со стоимостью разработки. Для программного обеспечения встроенных систем реального времени затраты на сопровождение могут в четыре раза превышать стоимость самой разработки. Высокие требования в отношении производительности и надежности таких программ предполагают их жесткую структуру, которая труднее поддается модификации.

Связывая сопровождение программ с характеристиками языка программирования, следует выделить, прежде всего, зависимость от читабельности. Обычно сопровождение выполняется не авторами программы, а другими лицами. В силу этого, плохая читабельность может крайне усложнить задачу усовершенствования и расширения функций программы.

Способы построения критериев эффективности

Возможны следующие способы построения критериев из частных показателей.

Выделение главного показателя. Из совокупности частных показателей $A_1, A_2, ..., A_n$ выделяется один, например A_1, который принимается за главный. На остальные показатели накладываются ограничения:

$$A_i \leqslant A_{i\text{доп}} \ (i = 2, 3, ..., n),$$

где $A_{i\text{доп}}$ — допустимое значение i-го показателя. Например, если в качестве A_1 выбирается легкость создания программ W (*Writability*), а на показатели надежности P и стоимости S накладываются ограничения, то критерий эффективности ЯП принимает вид:

$$W \to \max, P \leqslant P_{\text{доп}}, S \leqslant S_{\text{доп}}.$$

Способ последовательных уступок. Все частные показатели нумеруются в порядке их важности: наиболее существенным считается показатель A_1, а наименее важным — A_n. Находится минимальное значение показателя A_1 — $\min A_1$ (если нужно найти максимум, то достаточно изменить знак показателя). Затем делается «уступка» первому показателю ΔA_1, и получается ограничение $\min A_1 + \Delta A_1$.

На втором шаге отыскивается $\min A_2$ при ограничении $A_1 \leq \min A_1 + \Delta A_1$. После этого выбирается «уступка» для A_2: $\min A_2 + \Delta A_2$. На третьем шаге отыскивается $\min A_3$ при ограничениях $A_1 \leq \min A_1 + \Delta A_1$; $A_2 \leq \min A_2 + \Delta A_2$ и т. д. На последнем шаге ищут $\min A_n$ при ограничениях

$$A_1 \leq \min A_1 + \Delta A_1;$$

$$A_2 \leq \min A_2 + \Delta A_2;$$

$$\ldots$$

$$A_{n-1} \leq \min A_{n-1} + \Delta A_{n-1}.$$

Полученный на этом шаге вариант языка программирования и значения его показателей $A_1, A_2, ..., A_n$ считаются окончательными. Недостатком данного способа (критерия) является неоднозначность выбора ΔA_i.

Отношение частных показателей. В этом случае критерий эффективности получают в виде

$$K_1 = \frac{A_1, A_2, ..., A_n}{B_1, B_2, ..., B_m} \to \max, \tag{1.1}$$

или в виде

$$K_2 = \frac{B_1, B_2, ..., B_m}{A_1, A_2, ..., A_n} \to \min, \tag{1.2}$$

где A_i ($i = 1, 2, ..., n$) — частные показатели, для которых желательно увеличение численных значений, а B_i ($i = 1, 2, ..., m$) — частные показатели, численные значения которых нужно уменьшить. В частном случае критерий может быть представлен в виде

$$K_3 = \frac{B_1}{A_1} \to \min. \tag{1.3}$$

Возможной формой выражения (1.3) является критерий цены создания программы

$$K_4 = \frac{S}{W} \to \min, \tag{1.4}$$

где S — стоимость, W — легкость создания программы ЯП. Формула критерия K_4 характеризует затраты стоимости, приходящиеся на единицу легкости создания программы.

Аддитивная форма. Критерий эффективности имеет вид

$$K_5 = \sum_{i=1}^{n} \alpha_i A_i \to \max, \tag{1.5}$$

где $\alpha_1, \alpha_2, ..., \alpha_n$ — положительные и отрицательные весовые коэффициенты частных показателей. Положительные коэффициенты ставятся при тех показателях, которые желательно максимизировать, а отрицательные — при тех, которые желательно минимизировать.

Весовые коэффициенты могут быть определены методом экспертных оценок. Обычно они удовлетворяют условиям

$$0 \le \alpha_i < 1, \ \sum_{i=1}^{n} \alpha_i = 1. \tag{1.6}$$

Основной недостаток критерия заключается в возможности взаимной компенсации частных показателей.

Мультипликативная форма. Критерий эффективности имеет вид

$$K_6 = \prod_{i=1}^{n} A_i^{\alpha_i} \to \max, \tag{1.7}$$

где, в частном случае, коэффициенты α_i полагают равными единице.

От мультипликативной формы можно перейти к аддитивной, используя выражение:

$$\lg K_6 = \sum_{i=1}^{n} \alpha_i \lg A_i. \tag{1.8}$$

Критерий K_6 имеет тот же недостаток, что и критерий K_5.

Максиминная форма. Критерий эффективности описывается выражением:

$$K_6 = \prod_{i=1}^{n} A_i^{\alpha_i} \to \max, \tag{1.9}$$

Здесь реализована идея равномерного повышения уровня всех показателей за счет максимального «подтягивания» наихудшего из показателей (имеющего минимальное значение).

У максиминного критерия нет того недостатка, который присущ мультипликативному и аддитивному критериям.

Нормализация частных показателей

Частные показатели качества обычно имеют различную физическую природу и различные масштабы измерений, из-за чего их простое сравнение становится практически невозможным. Поэтому появляется задача приведения частных показателей к единому масштабу измерений, то есть их нормализация.

Рассмотрим отдельные способы нормализации.

Использование отклонения частного показателя от максимального.

$$\Delta A_i = A_{\max_i} - A_i. \tag{1.10}$$

В данном случае переходят к отклонениям показателей, однако способ не устраняет различия масштабов отклонений.

Использование безразмерной величины $\overline{A_i}$.

$$\overline{A_i} = \frac{A_{\max_i} - A_i}{A_{\max_i}}, \tag{1.11}$$

$$\overline{A_i} = \frac{A_i}{A_{\max_i}}. \tag{1.12}$$

Формула (1.11) применяется тогда, когда уменьшение A_i приводит к увеличению (улучшению) значения аддитивной формулы критерия. Выражение (1.12) используется, когда к увеличению значения аддитивной формулы критерия приводит увеличение A_i.

Учет приоритета частных показателей

Необходимость в учете приоритетов возникает в случае, когда частные показатели имеют различную степень важности.

Приоритет частных показателей задается с помощью ряда приоритета I, вектора приоритета $(b_1, ..., b_q, ..., b_n)$ и вектора весовых коэффициентов $(\alpha_1, \alpha_2, ..., \alpha_n)$.

Ряд приоритета представляет собой упорядоченное множество индексов частных показателей $I = (1, 2, ..., n)$. Он отражает чисто качественные отношения доминирования показателей, а именно отношения следующего типа: показатель A_1 важнее показателя A_2, а показатель A_2 важнее показателя A_3 и т. д.

Элемент b_q вектора приоритета показывает, во сколько раз показатель A_q важнее показателя A_{q+1} (здесь A_q — показатель, которому отведен номер q в ряду приоритета). Если A_q и A_{q+1} имеют одинаковый ранг, то $b_q = 1$. Для удобства принимают $b_n = 1$.

Компоненты векторов приоритета и весовых коэффициентов связаны между собой следующим отношением:

$$b_q = \frac{\alpha_q}{\alpha_{q+1}}.$$

Зависимость, позволяющая по известным значениям b_i определить величину α_q, имеет вид:

$$\alpha_q = \frac{\prod_{i=q}^{n} b_i}{\sum_{q=1}^{n}\prod_{i=q}^{n} b_i}.$$

Знание весовых коэффициентов позволяет учесть приоритет частных показателей.

Заключительные замечания

Когда собираются сторонники различных языков, обсуждение обычно принимает острую форму в стиле спора известных литературных героев Шуры Балаганова и Михаила Паниковского, страсти кипят, в выражениях не стесняются. Ведь известно, что профессионалы-программисты (в подавляющем большинстве) являются однолюбами-консерваторами, строящими храм для любимого языка и нещадно критикующими все остальные языки. Может быть, по этой причине большинство частных показателей качества ЯП, в частности читабельность, легкость создания и надежность, в настоящее время не являются ни строго определенными, ни точно измеримыми. Тем не менее предложенные схемы построения критериев вполне реальны и позволяют перевести дискуссию из эмоциональной в количественную плоскость. Нужно лишь заручиться поддержкой экспертов.

Конечно, здесь есть определенные трудности. Отдельные показатели имеют разный вес при рассмотрении языка с различных точек зрения. Разработчиков трансляторов в первую очередь интересует сложность реализации конструкций и структур данных языка. Программистов прежде всего волнует легкость создания программных приложений, а инженеров сопровождения — читабельность эксплуатируемых программ. Авторы языков программирования придают особое значение элегантности и возможности широкого использования языка. Мало того, некоторые характеристики, формирующие показатели, входят в противоречия друг с другом. Но все эти препятствия преодолимы при наличии доброй воли.

Контрольные вопросы и упражнения

1. Приведите три определения понятия «язык программирования». Чем они схожи? Чем отличаются?
2. Сравните понятия «знаковая ситуация» и «знаковая система».
3. В чем суть понятия «аппарат абстракции-конкретизации»?
4. Приведите примеры программных знаков и денотатов.
5. Что такое синтаксис языка программирования? Что он задает?
6. Что такое семантика языка программирования? Что она задает?

7. Какие категории ошибок сообщений в ЯП вы знаете? Прокомментируйте трудности их устранения.
8. Почему создание программ стоит так дорого?
9. Чему способствуют критерии эффективности, для чего и как они применяются?
10. Чем отличается понятие «показатель» от понятия «критерий»? Как они взаимодействуют друг с другом?
11. Приведите примеры противоречивых показателей. Как преодолеваются эти противоречия?
12. Если показателей качества много и они имеют для вас разную важность, как следует учитывать их влияние? Приведите конкретные примеры.

Глава 2

Виртуальные машины и трансляция языков

В данной главе материал посвящен обсуждению двух вопросов: среды, в которой выполняется программа, и процессу перевода текста программы в исполняемую форму. Изложение начинается с изучения аппаратной организации компьютеров, где выделяется принцип программного управления и рассматривается реализующая его структура, а затем описывается порядок функционирования компьютера. Вводится понятие виртуальной машины, устраняющей разрыв между языком, который «понимает аппаратура», и языком программирования, на котором написана программа. Далее это понятие распространяется на иерархию виртуальных машин, упрощающую взаимодействие между аппаратным уровнем и программными уровнями компьютерной системы. После чего объясняются возможности и категории специальных программ-переводчиков (компиляторов и интерпретаторов), занимающихся трансляцией — переводом текстов программ в форму, воспринимаемую аппаратурой компьютера. Дается развернутая характеристика всех этапов трансляции: лексического анализа, синтаксического анализа, семантического анализа, оптимизации, генерации кода и редактирования связей.

Аппаратная организация компьютеров

На аппаратном уровне компьютер или вычислительная машина (ВМ) является совокупностью технических средств, служащих для автоматизированной обработки дискретных данных по заданному алгоритму.

Алгоритм — одно из фундаментальных понятий математики и вычислительной техники. Международная организация стандартов (ISO) формулирует понятие *алгоритм* как «конечный упорядоченный набор четко определенных правил для решения проблемы» (ISO 2382/1-93). Помимо этой стандартизированной формулировки существуют и другие определения. Приведем наиболее распространенное из них: алгоритм — это точное предписание исполнителю, определяющее содержание и порядок действий, которые необходимо выполнить над исходными и промежуточными данными для получения конечного результата [28].

Запись алгоритма распадается на отдельные указания исполнителю — выполнить конкретное действие. Эти указания формулируются на языке, понятном

исполнителю. Указания алгоритма выполняются одно за другим. На каждом шаге исполнения алгоритма исполнителю известно — какое указание алгоритма должно выполняться следующим.

Таким образом, алгоритм — строгая, математическая форма описания метода решения задач.

Основными свойствами алгоритма являются: дискретность, определенность, массовость и результативность.

Дискретность выражается в том, что алгоритм описывает действия над дискретной информацией (например, числовой или символьной), причем сами эти действия также дискретны.

Свойство *определенности* означает, что в алгоритме указано все, что должно быть сделано, причем ни одно из действий не должно трактоваться двояко.

Массовость алгоритма подразумевает его применимость к множеству значений исходных данных, а не только к каким-то уникальным значениям.

Наконец, *результативность* алгоритма состоит в возможности получения результата за конечное число шагов.

Рассмотренные свойства алгоритмов предопределяют возможность их реализации на виртуальной машине, при этом процесс, порождаемый алгоритмом, называют *вычислительным процессом*.

Принцип программного управления

Вычислительная машина является исполнителем алгоритмов, поэтому именно свойства алгоритмов предопределяют ее организацию. Современные виртуальные машины построены на основе принципа программного управления. Основные идеи программного управления были изложены английским математиком Чарльзом Беббиджем (1883). Универсальную формулировку принципа программного управления предложил американский ученый Джон фон Нейман (1945). Рассмотрим его содержание.

1. Обрабатываемая информация кодируется двоичными цифрами (0, 1) и разделяется на единицы, называемые словами. Слово обрабатывается в виртуальной машине как единое целое (машинная единица информации).

   Алгоритм вычислений представляется в виртуальной машине в машинной форме — в форме программы, состоящей из последовательности *команд*. Команды тоже записываются в двоичном виде. Каждая команда предписывает некоторую операцию (из набора операций вычислительной машины) и указывает слова данных (числа), над которыми ее нужно выполнить.

   Операция задается значением кода операции КОП, а числа — адресами ячеек памяти $А_{чi}$, в которых они хранятся. Адрес $А_{ч}$ является машинным именем числа Ч. Адрес — единственное средство, с помощью которого можно найти нужное число в памяти. В типовой команде (рис. 2.1) $А_{ч1}$, $А_{ч2}$ обозначают адреса аргументов, а $А_{ч3}$ — адрес результата операции.

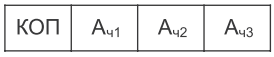


Рис. 2.1. Структура типовой команды

2. Команды программы хранятся в последовательности смежных ячеек памяти вычислительной машины и выполняются *в естественном порядке*, то есть в порядке их расположения в программе. При необходимости, с помощью специальных команд, естественный порядок выполнения может быть изменен. Решение об изменении порядка выполнения команд принимается самой программой, либо на основании анализа результатов предшествующих вычислений, либо безусловно. Важно уяснить, что вычисления в виртуальной машине определяются программой. Именно программа настраивает виртуальную машину на получение требуемых результатов. Замена программы приводит к изменению функций, реализуемых виртуальной машиной.
3. Команды и данные хранятся в одной и той же памяти, и внешне в памяти они неразличимы. Распознать их можно только по способу использования. Отыскиваются команды по адресам: чтобы найти команду K, надо знать адрес ячейки $A_к$, где она находится. Вводят программу в память так же, как и исходные данные.

Структура виртуальной машины

Большинство современных виртуальных машин по своей структуре отвечают принципу программного управления. Типичная фон-неймановская виртуальная машина (рис. 2.2) содержит: память, устройство ввода-вывода и процессор.

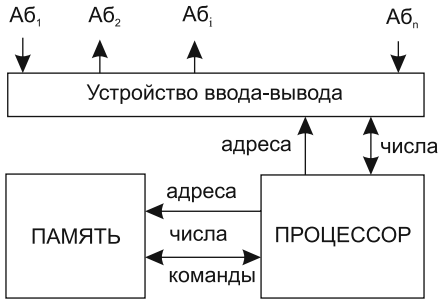


Рис. 2.2. Структура фон-неймановской вычислительной машины

Устройство ввода-вывода (УВВ) обеспечивает связь виртуальной машины с внешним миром. Все внешние источники и потребители информации называются абонентами виртуальной машины. Каждому абоненту присваивается свой адрес $Аб_1$, $Аб_2$, ..., $Аб_n$. По этому адресу виртуальная машина находит нужный абонент для обмена информацией. Абоненты отличаются друг от друга как скоростью работы, так и формой передаваемой (принимаемой) информации. А в виртуальной машине обрабатываются только двоичные коды, причем с постоянной скоростью. Возникает задача согласования формы изменения информации абонентов с аналогичными параметрами виртуальной машины. Эта задача решается устройством ввода-вывода. Таким образом, УВВ играет роль «электронного переводчика».

Память компьютера имеет сложную многоуровневую структуру, реализованную в виде взаимодействующих запоминающих устройств (ЗУ), которые могут использовать различные физические принципы для хранения данных.

Введенная информация сначала запоминается в основной памяти, а затем переносится во вторичную память для длительного хранения. Чтобы программа могла выполняться, команды и данные должны располагаться в *основной памяти* (ОП), организованной таким образом, что каждое двоичное слово хранится в отдельной ячейке, идентифицируемой адресом, причем соседние ячейки памяти имеют следующие по порядку адреса. Доступ к любым ячейкам основной памяти может производиться в произвольном порядке. Такой вид памяти известен как *память с произвольным доступом*. ОП современных виртуальных машинах в основном состоит из полупроводниковых *оперативных запоминающих устройств* (ОЗУ), обеспечивающих как считывание, так и запись информации. Для таких ЗУ характерна энергозависимость — хранимая информация теряется при отключении электропитания. Если необходимо, чтобы часть основной памяти была энергонезависимой, в состав ОП включают *постоянные запоминающие устройства* (ПЗУ), также обеспечивающие произвольный доступ. Хранящаяся в ПЗУ информация может только считываться.

Для долговременного хранения больших программ и массивов данных в виртуальной машине обычно имеется дополнительная память, известная как *вторичная*. Вторичная память энергонезависима и чаще всего реализуется на базе магнитных дисков. Информация в ней хранится в виде специальных программно поддерживаемых объектов — *файлов* (согласно стандарту ISO, файл — это именуемый набор записей, обрабатываемых как единый блок).

Неотъемлемой частью современных виртуальной машины стала *кэш-память* — память небольшой емкости, но высокого быстродействия. В нее из основной памяти копируются наиболее часто используемые команды и данные. При обращении со стороны процессора информация берется не из основной памяти, а из соответствующей копии, находящейся в более быстродействующей кэш-памяти.

Наконец, центральным устройством виртуальной машины является процессор. Процессор интерпретирует программу (истолковывает, раскрывает ее смысл) и на ее основе управляет работой всех устройств виртуальной машины, инициируя выполнение действий в памяти и УВВ. Функцией процессора является выборка команд из ОП и выполнение действий, предписанных командами. Говорят, что процессор является *аппаратным интерпретатором* команд.

При функционировании процессор обращается к памяти и одновременно посылает адрес ячейки, из которой нужно извлечь число (команду) или в которую нужно записать число. К внешним абонентам, как и к ячейкам памяти, процессор обращается по адресам через УВВ. При этом осуществляется соединение процессора с конкретным абонентом и производится обмен информацией.

Порядок функционирования виртуальной машины

Вычислительная машина работает циклически, выполняя в автоматическом режиме одну команду за другой. Рассмотрим порядок выполнения типовой команды.

1. *Чтение команды.* Адрес ячейки с командой $A_к$ известен (вначале это адрес ячейки $A_{к1}$ с первой командой). Процессор (ЦП) посылает его в память и получает оттуда команду, которая размещена в ячейке с указанным адресом

$$ЦП \rightarrow A_к \rightarrow ОП \rightarrow К \rightarrow ЦП$$

2. *Расшифровка кода операции*. Необходимое действие указывается в коде операции. Процессор расшифровывает код операции и предопределяет наименование текущей операции

   ```
   Наимен_операции = decod К(КОП)
   ```

3. *Выборка чисел (операндов)*. Допустим, что команда указала операцию: сложить. Для ее выполнения процессор начинает выбирать из памяти операнды — слагаемые. С этой целью он выделяет в команде адрес первого числа $А_{ч1}$ и посылает его в память. Из ячейки памяти с этим адресом выбирается слово и принимается в процессор.

   $ЦП \rightarrow А_{ч1} \rightarrow ОП \rightarrow Ч_1 \rightarrow ЦП$

   Точно так же из ячейки с адресом $А_{ч2}$, указанным в команде, читается второе число: второе слагаемое

   $ЦП \rightarrow А_{ч2} \rightarrow ОП \rightarrow Ч_2 \rightarrow ЦП$

4. *Выполнение операции*. Исходное состояние: числа (слагаемые) находятся в процессоре. В соответствии с кодом операции процессор выполняет указанную операцию (сложение, вычитание и т. д.) и определяет результат

   $Ч_1 + Ч_2 \rightarrow Р$

5. *Запись результата*. Результат операции, сформированный процессором, записывается в память. Для этого процессор выделяет в команде адрес $А_{ч3}$ и посылает результат в память по этому адресу

   $ЦП \rightarrow А_{ч3} \rightarrow ОП,\ ЦП \rightarrow Р \rightarrow ОП$

6. *Определение адреса следующей команды*. Обычно следующая команда находится в следующей ячейке. Ее адрес на единицу больше адреса текущей команды. Поэтому формирование адреса будет выполняться по выражению

   $А_к + 1 \rightarrow А_к$

После вычисления адреса очередной команды процессор приступает к ее выполнению, то есть переходит к пункту 1 описанной последовательности действий. Цикл выполнения все новых и новых команд повторяется до тех пор, пока процессор не выберет команду останова. В этом и состоит суть аппаратной интерпретации команд.

Понятие виртуальной машины

В совокупности команды аппаратного процессора составляют язык, на котором люди могут давать задания компьютеру. Такой язык называется встроенным *машинным языком* (МЯ). Конечно, состав команд машинного языка зависит от назначения компьютера, но в целом эти команды очень просты. Обычно их образуют команды пересылки данных, арифметической и логической обработки, ввода/вывода и управления потоком команд. Причем арифметическая обработка ограничивается сложением, вычитанием, умножением и делением. Примитивность машинных языков делает их использование трудным и утомительным.

Для преодоления сложности общения создают новые команды, более удобные для человека, чем машинные команды. Вместе эти новые команды образуют язык высокого уровня (ЯВУ). Но ведь подобный язык аппаратный компьютер «не понимает»! Существует два способа преодоления этого непонимания, их основная цель: заменять высокоуровневые команды эквивалентными наборами машинных команд. Правда, пути достижения этой цели используются разные.

Первый способ заключается в преобразовании одной программы, написанной на ЯВУ, в другую программу, записываемую в терминах машинного языка. Способ преобразования прост: каждая команда из ЯВУ заменяется на эквивалентный набор команд из машинного языка. Теперь аппаратный компьютер будет выполнять новую программу на машинном языке вместо исходной программы на ЯВУ. Исходная же программа на ЯВУ просто выбрасывается. Этот подход получил название *трансляции*.

Второй способ состоит в создании такой программы на машинном языке, которая поочередно обрабатывает каждую команду программы на языке высокого уровня: она заменяет каждую команду ЯВУ на эквивалентный набор машинных команд и сразу же выполняет этот набор. При этом новая программа на машинном языке не создается. Процесс «покомандного» перевода называют *интерпретацией*, а программу, выполняющую этот процесс, называют *интерпретатором*.

В конечном счете, и трансляция, и интерпретация являются инструментами поддержки такого важного понятия, как виртуальная машина [90]. *Виртуальная машина* — это программная надстройка над аппаратным компьютером. Виртуальная машина весьма удобна для программиста, поскольку в качестве машинного языка здесь выступает язык высокого уровня. Человек может считать, что ЯВУ встроен в виртуальную машину, и писать программы для машины в удобной для себя форме. Понятно, что реально всю работу по-прежнему будет выполнять аппаратный компьютер, обеспечиваемый транслятором или интерпретатором, но теперь это «остается за кадром».

Трансляторы и интерпретация

Современные технологии позволяют построить аппаратный компьютер, на котором непосредственно выполнялись бы программы, написанные на языке программирования высокого уровня. В итоге вы получили бы компьютер с таким встроенным машинным языком, как C++, Prolog, Haskell и т. д. Но экономическая целесообразность и эффективность такого компьютера крайне сомнительны. По эффективности, стоимости, быстродействию и гибкости компьютеры с низкоуровневыми машинными языками вне конкуренции. С другой стороны, в большинстве случаев программы пишутся на языках высокого уровня, существенно отличающихся от машинного языка. Перед исполнением на аппаратном компьютере такие программы подвергаются трансляции или интерпретации [1, 24, 46, 49].

Транслятором называют программное приложение, которое в качестве входных данных воспринимает программы на некотором *исходном языке*, а на выходе формирует эквивалентные по своей функциональности программы, но уже на другом, так называемом *объектном языке*. Как исходный язык, так и объектный язык может

быть высокого или низкого уровня. Охарактеризуем наиболее популярные разновидности трансляторов.

Ассемблером называют транслятор, у которого объектным языком является некоторая разновидность машинного языка какого-либо аппаратного компьютера, а исходным языком — символическое представление машинного языка. Исходный язык обычно называют *языком ассемблера*. Чаще всего каждая команда на исходном языке переводится в одну команду на объектном языке.

Компилятор — это транслятор, для которого исходным считается язык высокого уровня. Объектный язык очень близок к машинному языку аппаратного компьютера — им является либо язык ассемблера, либо какой-нибудь вариант машинного языка. Например, программы на языке C обычно компилируются в программы на языке ассемблера, которые ассемблер затем переводит в машинный язык.

Загрузчик или редактор связей — это транслятор, у которого объектный язык состоит из готовых к выполнению машинных команд, а исходный язык очень близок к объектному. Обычно исходный язык описывает программы на машинном языке, представляемые в перемещаемой форме, а также включает в себя таблицы данных. Таблицы данных отмечают те точки, в которых перемещаемый код должен быть изменен, чтобы стать действительно выполняемым. Например, некоторая программа X должна быть откомпилирована для размещения и использования в памяти по адресам от 0 до 100, а программа Y — по адресам от 0 до 1000. Кроме того, эти программы могут использовать библиотечные функции, для которых выделены адреса от 0 до 4000. Редактор связей должен создать единую выполняемую программу, в которой применяются согласованные адреса (табл. 2.1). Как видим, выполняемая программа скомпонована как единая программа с используемыми адресами от 0 до 5102.

Таблица 2.1. Распределение памяти редактором связей

Программа	Адреса после компиляции	Адреса в выполняемой программе
X	0–100	0–100
Y	0–1000	101–1101
Библиотечные функции	0–4000	1102–5102

Препроцессор, или *макропроцессор*, — это транслятор с исходным языком в виде расширенной формы некоторого языка программирования высокого уровня (C, C++ и т. д.) и объектным языком в виде стандартной версии этого языка. Препроцессор создает такую объектную программу, которая готова к вводу в транслятор исходного стандартного языка. В состав большинства компиляторов для языка C входит препроцессор, который до начала фазы компиляции преобразует макросы, записанные в программе, в стандартные операторы языка C.

Трансляция программ с исходного языка высокого уровня на машинный язык обычно выполняется за нескольких шагов. Например, типична следующая последовательность шагов:

1. Исходный текст программы на языке C++ транслируется в текст на C.
2. Текст программы на C компилируется в программу на языке ассемблера.

3. Редактор связей преобразует программу на языке ассемблера в выполняемый машинный код.
4. Загрузчик загружает в память выполняемый машинный код, который теперь может быть выполнен.

Мало того, сам процесс компиляции может делиться на несколько последовательных этапов, каждый из которых создает некоторую промежуточную форму программы, прежде чем сформируется конечная объектная программа.

Перечислим характерные недостатки механизма трансляции:

- потеря информации об исходной программе на языке высокого уровня. Если при выполнении объектной формы программы появляется ошибка, трудно выявить высокоуровневый оператор, являющийся ее источником;
- существенный рост размера объектной формы программы, поскольку оператор на языке высокого уровня содержит гораздо больше информации, чем команда машинного языка.

Помимо трансляции высокоуровневых программ в эквивалентные программы на машинном языке, можно использовать и другой подход, называемый *программной интерпретацией*. В этом случае создается виртуальная машина — виртуальный компьютер, для которого машинным языком будет некоторый язык высокого уровня. Виртуальный компьютер — это аппаратный компьютер + набор программ на машинном языке, которые моделируют алгоритмы и структуры данных, необходимые для выполнения программ на языке высокого уровня. Иными словами, здесь при помощи программ, написанных на машинном языке аппаратного компьютера, моделируется абстрактный компьютер с машинным языком высокого уровня. Впрочем, виртуальный компьютер можно разработать и аппаратным способом. Дело лишь в стоимости подобного решения. Для виртуального компьютера заданиями являются программы на языке высокого уровня. Основная моделирующая программа такого компьютера, используя некоторый алгоритм интерпретации, декодирует и выполняет один за другим каждый оператор программы пользователя, формируя необходимый результат. Применительно к конкретному виртуальному компьютеру трудно выделить: выполняется ли программа непосредственно аппаратными средствами или сначала ее элемент преобразуется в последовательность встроенных машинных команд, а затем уже выполняется.

Подчеркнем сходство и различия трансляции и интерпретации. В любом случае входными данными считаются программы на языке высокого уровня. Однако в результате трансляции создается новая программа на объектном языке, которая затем должна выполняться аппаратурой, а если более точно — аппаратным интерпретатором для этого объектного языка. С другой стороны, программный интерпретатор непосредственно выполняет программу пользователя. Трассировка производимых вычислений показывает:

- транслятор обрабатывает операторы программы в порядке их фактического ввода, причем каждый оператор обрабатывается только один раз;
- интерпретатор следует логике управления пользовательской программы и может обрабатывать некоторые операторы многократно (если они являются частью

цикла) или полностью игнорировать другие операторы (если на них не будет передано управление).

К достоинству интерпретации следует отнести тот факт, что операторы программы остаются в своей исходной форме до тех пор, пока они не понадобятся при выполнении. Отсюда вывод: не расходуется память на хранение нескольких копий длинной цепочки машинных команд; все необходимые команды достаточно сохранить в программе-интерпретаторе лишь один раз. Однако платой за это становится необходимость многократного декодирования одного и того же оператора цикла (или вызова подпрограмм). Для минимизации затрат к таким операторам следует применять механизм трансляции.

В чистом виде трансляция и интерпретация являются двумя крайностями. Обычно эти подходы применяют совместно, взаимно компенсируя недостатки друг друга. Чистая трансляция удобна, если исходный язык достаточно близок к машинному языку (например, это язык ассемблера). Чистая интерпретация целесообразна для интерактивных языков и для языков управления операционными системами. Реальное, комбинированное решение иллюстрируется рис. 2.3. Сначала независимые части программы транслируются в объектные формы (этап трансляции). На втором этапе, этапе загрузки, эти независимые части объединяются с набором подпрограмм поддержки выполнения. Подпрограммы поддержки обеспечивают программные реализации специфических операций, объединение с ними формирует выполняемую форму программы. Операторы этой формы декодируются и интерпретируются на третьем этапе, этапе выполнения.

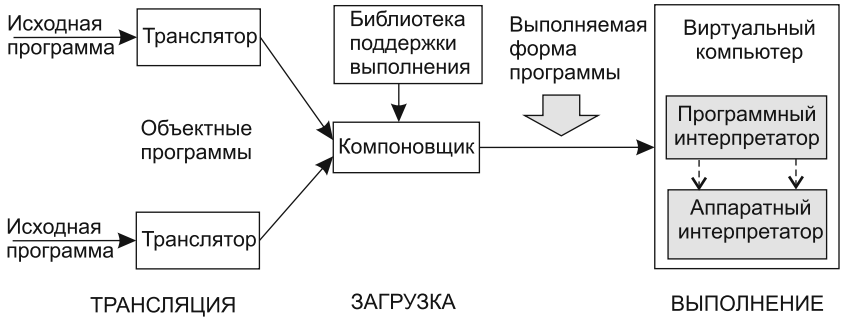


Рис. 2.3. Этапы обработки программы пользователя

С точки зрения реализации все языки программирования делятся на компилируемые и интерпретируемые.

Языки C, C++, C#, Fortran, Pascal и Ada относят к категории компилируемых языков: перед началом выполнения их программы транслируются в машинный код аппаратного компьютера. Программной интерпретации подвергаются лишь программы поддержки выполнения, которые моделируют элементарные операции языка программирования, не имеющие близкого аналога в машинном языке. Транслятор компилируемого языка — большое и сложное программное приложение, обеспечивающее создание программ, максимально эффективных по скорости выполнения.

Языки LISP, ML, Perl, Prolog и Smalltalk являются представителями категории интерпретируемых языков: здесь тоже используют транслятор, но он выдает не

машинный код аппаратного компьютера, а промежуточную форму программы. Эта форма отличается от встроенного машинного кода. Процесс интерпретации для выполнения промежуточной формы реализуется программным путем, поскольку аппаратный интерпретатор непосредственно не применим. К сожалению, программный интерпретатор существенно замедляет выполнение программы. Трансляторы интерпретируемых языков считаются довольно простыми программами, основная сложность кроется в программной поддержке процесса интерпретации.

Ориентация языка Java на веб-среду обусловила его реализацию в качестве интерпретируемого языка. Компилятор Java создает промежуточный набор *байт-кодов* для виртуальной машины Java. Преимущество такого решения заключается в том, что скомпилированный на одной машине байт-код может быть выполнен на другой, например, будучи передан по сети. Основной причиной неэффективности веб-приложений является потеря времени при передаче по сети затребованных пользователем страниц, а не выполнение программ на аппаратном компьютере клиента. Кроме того, веб-сервер не в состоянии предугадать машинную организацию компьютера клиента. Поэтому браузер и формирует виртуальную машину Java, исполняющую стандартный набор байт-кодов Java.

Иерархия виртуальных машин

На практике виртуальная машина, которую программист применяет для создания приложений, состоит из *иерархии виртуальных машин*. Фундаментом, основанием этой иерархии является, конечно, аппаратный, базовый компьютер. Впрочем, программист крайне редко взаимодействует именно с ним. Каждый слой программного обеспечения вносит свою лепту в изменение функционального образа базового компьютера, а итоговый образ, с учетом всей иерархии программных слоев, может существенно отличаться от аппаратного компьютера. Если считать первым слоем аппаратуру, то вторым слоем является *операционная система*, представляющая собой сложный комплекс программ [90].

С одной стороны, операционная система добавляет много новых операций и структур данных, которые не обеспечены аппаратными средствами компьютера (средства управления файлами, службу времени и т. д.). С другой стороны, из виртуальной машины, определяемой операционной системой, в целях безопасности исключаются и становятся недоступными для пользователя отдельные аппаратные операции (операции ввода-вывода, выключения компьютера, мультипрограммирования и организации мультипроцессорной работы). Обычно разработчик реализации для языка программирования имеет дело именно с такой виртуальной машиной, определяемой операционной системой. В процессе этой реализации разработчик формирует новый слой программного обеспечения, которое выполняется на двухслойной виртуальной машине и моделирует операции для языка высокого уровня, а также механизм трансляции.

Иерархия виртуальных машин может и не ограничиваться тремя описанными слоями. Например, новые программы способны наращивать количество слоев в этой иерархии. На рис. 2.4 представлена иерархия виртуальных машин с точки зрения пользователя веб-среды.

Рис. 2.4. Иерархия виртуальных машин для веб-среды

Над виртуальной машиной для языка C++ программист надстраивает программное приложение, выполняющее роль веб-браузера. Этот браузер создает виртуальную веб-машину, способную обрабатывать структуры данных в веб-среде, поддерживать гиперссылки для перехода к другим веб-сайтам, отображать веб-страницы, а также выполнять интерактивные программы для пользователей браузера.

Этапы трансляции

Процесс трансляции исходной программы в ее исполняемую форму может оказаться относительно простым, но чаще он достаточно сложен. При трансляции максимальная часть усилий направлена на создание эффективно выполняемого кода. Форма такого кода значительно сложнее структуры исходной программы.

В процесс трансляции выделяют две основные стадии: *анализ* исходной программы и *синтез* выполняемой объектной программы.

На стадии анализа исходная программа разбивается на составные части, на которые накладывается грамматическая структура. Затем эта структура используется для создания промежуточного представления исходной программы. Если обнаруживается, что исходная программа имеет синтаксические или семантические ошибки, пользователю выдаются сообщения об этом, чтобы он мог исправить найденные ошибки. При анализе также собирается информация об исходной программе, которая сохраняется в структуре данных, именуемой таблицей символов. Таблица символов вместе с промежуточным представлением программы передается на стадию синтеза.

На стадии синтеза на основе промежуточного представления и информации из таблицы символов строится объектный код программы. Анализ часто называют начальной стадией (front end), а синтез — заключительной (back end).

В большинстве трансляторов эти стадии не имеют четкой границы; чаще они бывают настолько взаимосвязаны, что анализ и синтез чередуются. На рис. 2.5 представлена структура типичного компилятора.

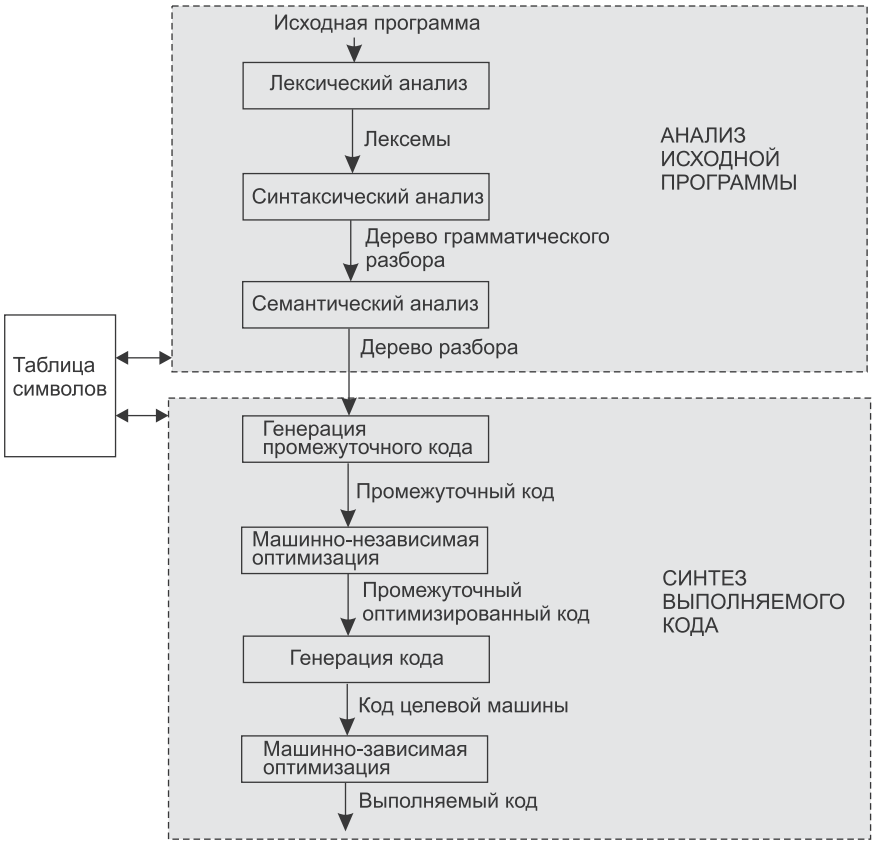


Рис. 2.5. Структура типичного компилятора

Некоторые компиляторы между анализом и синтезом содержат этап машинно-независимой оптимизации. Назначение этой оптимизации — преобразовать промежуточное представление, чтобы синтез мог получить более качественную объектную программу по сравнению с той, которая следует из неоптимизированного промежуточного представления. Поскольку оптимизация необязательна, некоторые этапы оптимизации, показанные на рис. 2.5, в компиляторе могут отсутствовать.

Трансляторы можно классифицировать по количеству *проходов*, которые они делают по тексту исходной программы. Как правило, простой компилятор использует два прохода. Во время первого прохода анализ программы приводит к ее декомпозиции на составляющие сегменты, а также к формированию информации, необходимой для следующего прохода (сведения об использовании имен переменных и т. д.). В ходе второго прохода обычно генерируется сама объектная программа.

В простейших случаях можно ориентироваться на единственный проход, при котором после анализа программа сразу же преобразуется в объектный код. Большинство компиляторов для языка Pascal работают по однопроходной схеме. Для оптимизации программ по скорости выполнения применяют многопроходные схемы компиляции:

- В первом проходе осуществляется анализ исходной программы.
- Во втором проходе по алгоритму оптимизации создается более эффективная форма программы.
- В третьем проходе генерируется объектный код.

Впрочем, практика показывает, что решающим фактором здесь оказывается сложность самого языка программирования, а не количество проходов компилятора.

Анализ исходной программы

С точки зрения транслятора исходная программа — это длинная цепочка, состоящая из многих тысяч символов. Транслятор не рассматривает такие элементы программы, как процедуры, операторы, объявления; он выполняет анализ программы, разбирая ее текст последовательно, символ за символом.

Лексический анализ, или сканирование. Первый этап компиляции называется *лексическим анализом*, или *сканированием*. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами. Для каждой лексемы анализатор строит выходной токен (token) вида

```
<имя_токена, значение_атрибута>
```

Он передается последующему этапу, синтаксическому анализу. Первый компонент токена, `имя_токена`, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице символов, соответствующую данному токену. Информация из записи в таблице символов необходима для семантического анализа и генерации кода.

Предположим, например, что исходная программа содержит оператор присваивания

```
area = width + height * 17
```

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены, передаваемые синтаксическому анализатору:

1. `area` представляет собой лексему, которая может отображаться в токен `<id, 1>`, где `id` — абстрактный символ, обозначающий идентификатор, а `1` указывает запись в таблице символов для `area`. Запись таблицы символов для некоторого идентификатора хранит информацию о нем, такую как его имя и тип.
2. Символ присваивания `=` представляет собой лексему, которая отображается в токен `<=>`. Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например такой, как `assign`, но для удобства записи мы будем использовать в качестве имени абстрактного символа саму лексему.
3. `width` представляет собой лексему, которая отображается в токен `<id, 2>`, где `2` указывает на запись в таблице символов для `width`.
4. `+` является лексемой, отображаемой в токен `<+>`.
5. `height` — лексема, отображаемая в токен `<id, 3>`, где `3` указывает на запись в таблице символов для `height`.

6. * — лексема, отображаемая в токен <*>.
7. 17 — лексема, отображаемая в токен <17>.

Пробелы, разделяющие лексемы, лексическим анализатором отбрасываются.

На рис. 2.6 показано представление оператора присваивания после лексического анализа в виде последовательности токенов

```
id1 <=> id2 <+> id3 <*> <17>
```

При этом представлении имена токенов =, + и * представляют собой абстрактные символы для операций присваивания, сложения и умножения соответственно.

Синтаксический анализ, или разбор (parsing). Вторым этапом трансляции является *синтаксический анализ*, или *разбор*. На этом этапе лексемы (токены), являющиеся результатами лексического анализа, применяются для выявления более крупных программных структур: операторов, объявлений, выражений и т. п. Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением считается синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы — аргументы этой операции. Синтаксическое дерево для потока токенов показано на выходе синтаксического анализатора на рис. 2.6.

Это дерево указывает порядок выполнения операций в операторе присваивания

```
area = width + height * 17
```

Дерево имеет внутренний узел, помеченный *, левым дочерним узлом которого является <id, 3>, а правым — 17. Узел <id, 3> представляет идентификатор height. Узел, помеченный *, явно указывает, что сначала мы должны умножить значение height на 17. Узел, помеченный +, указывает, что мы должны прибавить результат умножения к значению width. Корень дерева с меткой = говорит о том, что следует присвоить результат этого сложения ячейке памяти с идентификатором area. Порядок операций согласуется с обычными арифметическими правилами, которые говорят о том, что умножение имеет более высокий приоритет, чем сложение, и должно быть выполнено до сложения.

Последующие фазы компилятора используют грамматическую структуру, которая помогает проанализировать исходную и сгенерировать объектную программу.

Семантический анализ. Семантический анализ считается самым важным этапом трансляции. На этом этапе обрабатываются структуры, выявленные синтаксическим анализатором, начинает формироваться структура выполняемого объектного кода. По сути, семантический анализ играет роль моста, соединяющего две стадии трансляции — анализ и синтез. Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом; компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

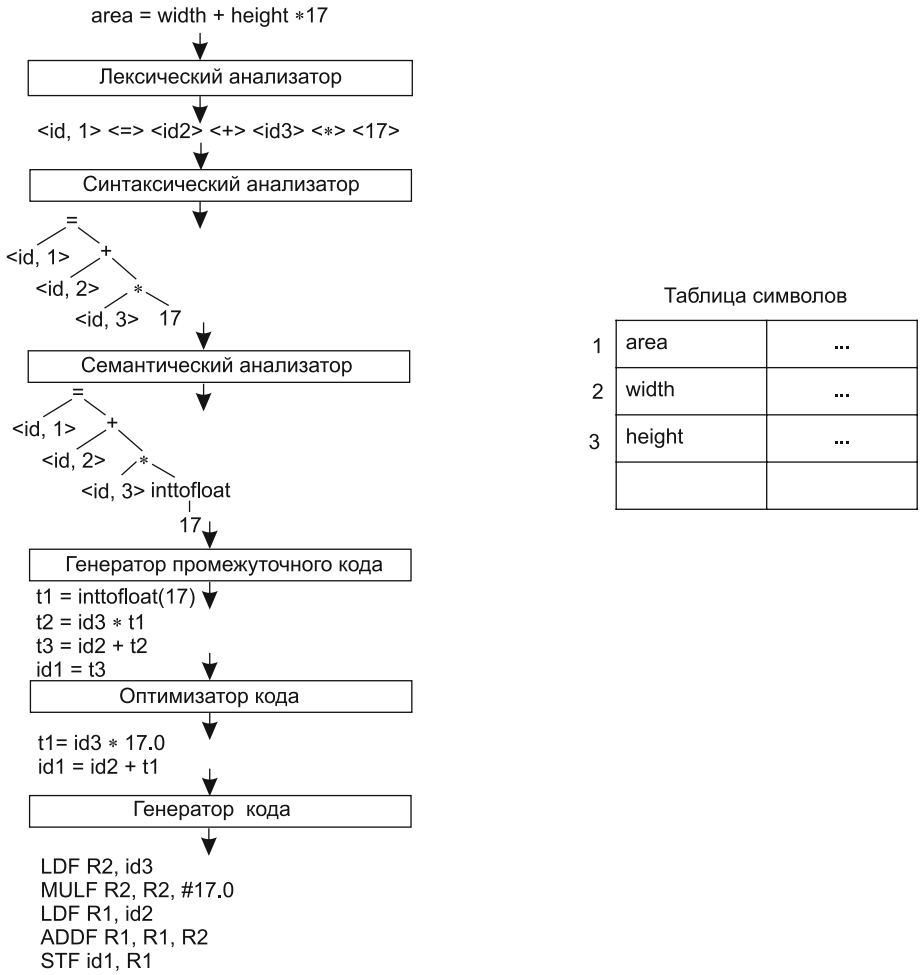


Рис. 2.6. Трансляция оператора присваивания

Спецификация языка может разрешать неявные преобразования типов, именуемые неявными приведениями (coercion). Например, бинарная арифметическая операция может быть применена либо к паре целых чисел, либо к паре чисел с плавающей точкой. Если такая операция применена к числу с плавающей точкой и целому числу, то компилятор может выполнить преобразование целого числа в число с плавающей точкой.

Такое неявное приведение показано на рис. 2.6. Предположим, что `area`, `width` и `height` были объявлены как числа с плавающей точкой и что лексема 17 образует целое число. Проверка типов в семантическом анализаторе на рис. 2.6 определяет, что операция ∗ применяется к числу с плавающей точкой `height` и целому числу 17. В этом случае целое число может быть преобразовано в число с плавающей точкой. Обратите внимание, что на рис. 2.6 в синтаксическом дереве, полученном на выходе семантического анализатора, имеется дополнительный узел для оператора

`inttofloat`, который явным образом преобразует свой целый аргумент в число с плавающей точкой.

В ходе семантического анализа выполняется и ряд других функций: поддержка таблицы символов, обнаружение большинства ошибок, замена макросов их определениями и выполнение операторов времени компиляции. В простейшем случае семантический анализатор может сформировать выполняемый объектный код, но чаще всего он создает некую внутреннюю форму, которая подвергается оптимизации, и лишь затем генерируется окончательный выполняемый код.

В действительности семантический анализатор образует набор специализированных анализаторов, каждый из которых обрабатывает некоторую программную конструкцию. Эти анализаторы взаимодействуют между собой при помощи информации из таблицы символов. Например, обработчик объявлений переменных заносит объявленные типы в таблицу символов. Обработчик арифметических выражений использует описанные типы для генерации объектного кода операций. Опишем наиболее общие функции специализированных анализаторов.

Поддержка таблицы символов. Таблица символов имеет сложную структуру. В таблице символов содержатся не только имена, но и другие, самые разнообразные атрибуты идентификаторов:

- вид (простая переменная, имя массива, имя подпрограммы, формальный параметр и т. д.);
- тип значения (целочисленный, вещественный и т. д.);
- адресные ссылки;
- любые сведения, извлекаемые из объявления.

Все эти данные надо сохранять и использовать. Обычно таблица символов уничтожается после окончания трансляции, но в отдельных случаях она может сохраняться и в период выполнения программы.

Включение неявной информации. Достаточно часто неявную информацию из исходной программы нужно отразить в объектной программе самым явным образом. Как правило, это относится к *соглашениям по умолчанию*. Например, если в языке Fortran тип переменной в программе не задан объявлением, то по умолчанию этой переменной приписывается тип, зависящий от начального символа в ее имени.

Обнаружение ошибок. На различных этапах трансляции возможно появление самых разнообразных ошибок, например:

- разделитель операторов в середине выражения;
- объявление среди последовательности операторов;
- символ операции вместо ожидаемого идентификатора;
- вещественная переменная вместо ожидаемой целой переменной;
- индексированная переменная с двумя индексами вместо элемента одномерного массива.

Анализатор должен не только распознавать ошибки и выдавать сообщения о них, но и принимать разумное решение о продолжении трансляции.

Макрообработка и служебные операции. Макросом называют такую часть текста программы, которая определена отдельно и должна быть вставлена в программу во

время трансляции, если в программе она вызывается. Служебные операции обеспечивают контроль над трансляцией исходной программы и выполняются в ходе компиляции. Например, в языке C имеется несколько таких операций:

- Операция `#define` позволяет вычислять значение констант и выражений до начала компиляции программы.
- Операция `#ifdef` (if-defined) позволяет транслировать какой-то один из нескольких альтернативных фрагментов кода (в зависимости от наличия или отсутствия определенных переменных).

Синтез объектной программы

На заключительной стадии трансляции создается выполняемая форма программы на основе того, что было сделано семантическим анализатором. На этой стадии обязательно генерируется код, а дополнительно может осуществляться и оптимизация программы. Если части программы транслировались отдельно или использовались библиотечные программы, то для получения полностью пригодной к выполнению программы необходимы редактирование связей и загрузка.

Генерация промежуточного кода. В процессе трансляции исходной программы в целевой код компилятор может создавать одно или несколько промежуточных представлений различного вида. Синтаксические деревья являются видом промежуточного представления; обычно они используются в процессе синтаксического и семантического анализа.

После синтаксического и семантического анализа исходной программы многие компиляторы генерируют явное низкоуровневое или машинное промежуточное представление исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины. Такое промежуточное представление должно обладать двумя важными свойствами: оно должно легко генерироваться и легко транслироваться в целевой машинный язык.

Выход генератора промежуточного кода на рис. 2.6 состоит из последовательности кодов, адресность которых колеблется от 1 до 3:

```
t1 = inttofloat(17)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Оптимизация кода. Этап машинно-независимой оптимизации кода пытается улучшить промежуточный код, чтобы затем получить более качественный объектный код. Обычно «более качественный», «лучший» означает «более быстрый», но могут применяться и другие критерии оптимизации, как, например, «более короткий код» или «код, использующий меньшее количество ресурсов». Например, как вы видели, непосредственный алгоритм создает четыре строчки промежуточного кода, используя по команде для каждого оператора в синтаксическом дереве, полученном на выходе семантического анализатора.

Оптимизация заключается в поиске рационального способа генерации хорошего целевого кода. Оптимизатор может определить, что преобразование `17` из целого числа в число с плавающей точкой может быть выполнено единственный раз во

время компиляции, так что операция `inttofloat` может быть устранена путем замены целого числа `17` числом с плавающей точкой `17.0`. Кроме того, `t3` используется только один раз — для передачи значения в `id1`, так что оптимизатор может преобразовать четыре команды в более короткую последовательность из двух команд:

```
t1 = id3 * 17.0
id1 = id2 + t1
```

Спектр применяемых на этом этапе методов очень широк. Некоторые компиляторы анализируют программу для выявления других возможностей оптимизации:

- однократное вычисление общих подвыражений;
- вынесение инвариантных операций из тела цикла;
- оптимизация использования регистров;
- оптимизация вычисления формул доступа к элементам массива.

Так называемые «оптимизирующие компиляторы» затрачивают на этот этап достаточно много времени, в то время как другие компиляторы используют лишь простые методы оптимизации, которые существенно повышают скорость работы объектной программы, но не слишком замедляют процесс компиляции.

Генерация кода. Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательности машинных команд, выполняющих те же действия. Ключевым моментом генерации кода является аккуратное распределение регистров для хранения переменных.

Например, при использовании регистров `R1` и `R2` двухстрочный промежуточный код может транслироваться в следующий машинный код

```
LDF R2, id3
MULF R2, R2, #17.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Первый операнд каждой команды определяет приемник результата. `F` в коде операции команды говорит о том, что команда работает с числами с плавающей точкой. Записанный код загружает содержимое ячейки с адресом `id3` в регистр `R2`, затем умножает его на константу с плавающей точкой `17.0.` Символ `#` указывает, что `17.0` следует рассматривать как непосредственное значение. Третья команда помещает `id2` в регистр `R1`, а четвертая прибавляет к нему предварительно вычисленное и сохраненное в регистре `R2` значение. Наконец, значение регистра `R1` сохраняется по адресу `id1`, так что код корректно реализует оператор присваивания `area = width + height * 17`.

Для простоты изложения мы полностью игнорировали проблему распределения памяти под идентификаторы в исходной программе. Решения о распределении памяти принимаются либо в ходе генерации промежуточного кода, либо при генерации целевого кода.

Редактирование связей и загрузка. Этот заключительный этап трансляции востребован далеко не всегда. На данном этапе фрагменты раздельно откомпили-

рованного кода объединяются в единую выполняемую программу. Как правило, исполняемые программы, полученные на предыдущих этапах, почти готовы к использованию. Исключение составляют лишь адресные ссылки на внешние данные или другие программы. Связи между этими разрозненными фрагментами записаны в уже созданных *таблицах загрузчика*. Загрузчик (редактор связей) загружает различные фрагменты кода в память, а затем использует таблицы загрузчика для связывания фрагментов в единую программу. В ходе связывания в код добавляются необходимые адресные ссылки, а иногда и данные.

Контрольные вопросы

1. Исследуйте реализацию хорошо известного вам языка программирования. Что является исполняемой формой программы — результатом работы транслятора? Какие этапы трансляции происходят при преобразовании различных выражений и операторов в их исполняемую форму? Какая программная интерпретация необходима при выполнении программы? Является ли интерпретатор программно моделируемым? Какие из примитивных операций требуют программной интерпретации?
2. На примере собственного компьютера определите структуру вашего виртуального компьютера на уровне операционной системы. Чем этот виртуальный компьютер отличается от аппаратного компьютера? Ограничивает ли операционная система использование аппаратных средств (например, существуют ли какие-либо машинные команды, которые становятся недоступными для использования в пользовательских программах)? Какие дополнительные возможности (по сравнению с аппаратным компьютером) предоставляет вам виртуальный компьютер?
3. Операционная система позволяет программисту применять виртуальный компьютер, который существенно отличается от базового аппаратного компьютера. Основные преимущества этого виртуального компьютера заключаются в следующем: 1) расширяются возможности ввода-вывода и упрощается его спецификация; 2) компьютер защищается от неумышленных ошибок пользователя, причем эти ошибки изолируются от других пользователей; 3) системные ресурсы эффективно распределяются между пользователями. Проанализируйте вашу операционную систему и предоставляемый вам виртуальный компьютер с точки зрения качества решения перечисленных задач. Выявите достоинства, а также недостатки операционной системы и виртуального компьютера.
4. Выделите 2–3 программы, которыми вы любите пользоваться на домашнем компьютере. Сравните предоставляемые ими виртуальные машины. Насколько сильно они отличаются от виртуальной машины уровня «операционная система»? Какие дополнительные возможности приносят? Каких возможностей лишают?
5. В чем заключается разница между компилятором и интерпретатором?
6. Каковы преимущества: а) компилятора перед интерпретатором и б) интерпретатора перед компилятором?

7. Каковы преимущества транслятора, которой на выходе дает код на языке ассемблера, по сравнению с транслятором, которой на выходе дает код на машинном языке?
8. Компилятор, который транслирует программу на высокоуровневом языке программирования в программу на другом высокоуровневом языке программирования, называется транслятором из исходного текста в исходный текст (source-to-source). Каковы преимущества использования языка программирования C в качестве целевого для такого компилятора?

Глава 3
Виды языков программирования

В этой главе обсуждаются различные разновидности языков программирования: императивные, функциональные, логические и объектно-ориентированные языки. По каждой разновидности приводится отдельное описание одного-двух языков. Кроме того, дается характеристика языка гипертекстовой разметки HTML. Самое пристальное внимание обращено на скриптовые языки. Помимо их общей характеристики включены справки по языкам Perl, JavaScript, PHP, Python, Ruby и Lua. Глава завершается рассмотрением таких гибридных языков разметки/программирования, как XSLT и JSP.

И наконец, заключительное замечание. Специфика главы вынуждает использовать в ней многие понятия, полное содержание которых станет ясным лишь после изучения последующих разделов книги. Там, где это возможно, пояснения будут даваться по ходу изложения (правда, в упрощенном виде). В любом случае, для получения полной картины к материалу данной главы имеет смысл еще раз вернуться после ознакомления со всем учебником.

Парадигмы программирования

Парадигма — это главная идея какого-либо сложного понятия (в данном случае языка программирования) [3, 13, 47].

Применительно к языкам программирования различают следующие парадигмы:

- императивное программирование;
- функциональное программирование;
- логическое программирование;
- объектно-ориентированное программирование.

В императивном программировании задание для компьютера формируется в виде последовательности команд, в функциональном программировании — указанием функций. В логическом программировании применяются высказывания, а в объектно-ориентированном программировании — объекты.

Каждую парадигму поддерживает свой вид языков программирования:

- *Императивные ЯП* ориентированы на операторы. Вычисления в них представляются как последовательность действий, производимых операторами.
- *Функциональные ЯП* задают вычисления как вызовы функций.

- *Логические ЯП* описывают вычисления с помощью формальной логики.
- В *объектно-ориентированных ЯП* вычисления реализуются совокупностью объектов.

Конечно, весьма наивно классифицировать языки программирования только по одному основанию — парадигме, поэтому в данной главе приведен обзор и языков разметки, и скриптовых языков.

Языки разметки, например HTML, не являются языками программирования. Эти языки не описывают вычислений, а всего лишь задают общий вид документа. Тем не менее к языкам разметки применимы критерии оценки, приведенные в первой главе. В частности, совершенно очевидна важность читабельности и легкости создания кодов разметки.

Императивные языки программирования

Императивные (процедурные) языки программирования (ЯП) задают вычисления как последовательность команд (операторов).

Они ориентированы на компьютеры с архитектурой фон Неймана.

Основные понятия императивных ЯП тесно связаны с компонентами компьютера:

- переменные различных типов (моделируют ячейки памяти);
- операторы присваивания (моделируют пересылки данных);
- повторения действий в форме итерации (моделируют хранение информации в смежных ячейках памяти).

При реализации оператора присваивания происходит следующее: операнды выражения передаются из памяти в процессор, а результат вычисления выражения заносится в ячейку памяти, именуемую левой частью оператора. Поскольку операторы хранятся в соседних ячейках памяти, то итерации на компьютере фон Неймана осуществляются очень быстро.

Вычисления основываются на понятии *состояние компьютера*. Состояние компьютера — это множество всех значений всех ячеек его памяти. Программа состоит из последовательности операторов, выполнение каждого из которых влечет за собой изменение значения в одной или нескольких ячейках памяти, то есть переход компьютера в новое состояние. В общем случае синтаксис императивной программы имеет вид:

```
оператор1;
оператор2;
…
```

Чаще всего операторы выполняются в порядке их следования в программе, друг за другом, и приводят к последовательной смене состояний компьютера. Конечное состояние обеспечивает требуемый результат. Словом, императивные программы гармонизированы с принципами работы традиционного компьютера, реализуются быстро и эффективно. Именно этим объясняется высокая популярность императивных ЯП.

Императивные языки образуют очень многочисленный отряд, приведем названия самых известных языков: Fortran (1954, Д. Бэкус), Algol 60 (1960, П. Наур), Cobol (1960, Пентагон), PL/1 (1965, IBM), Pascal (1971, Н. Вирт), C (1972, Д. Ритчи). В скобках указаны год и автор языка.

Язык Fortran

Fortran является одним из первых языков, который и сейчас широко применяется в области научных вычислений. За более чем пятидесятилетнюю историю этот язык претерпел многие изменения, его много раз называли устаревшим, но он продолжает существовать и развиваться [48].

Fortran — это первый язык программирования высокого уровня, который получил широкое признание. В своем исходном варианте Fortran был создан фирмой *IBM* в 1957 году. В то время программисты сомневались в целесообразности использования языков высокого уровня. Наиболее серьезным аргументом «против» была малая эффективность выполнения кода, получающегося после трансляции программ, написанных на этих языках. Поэтому первые версии языка были ориентированы на обеспечение эффективности выполнения. Первое стандартное определение языка появилось в 1966 году, а затем, в семидесятых годах, были внесены существенные изменения, которые привели к появлению языка Fortran 77. В XXI веке сформирована версия Fortran 2008.

Обычно при реализации языка Fortran используется стандартная технология трансляции. Для написания программы применяется простой текстовый редактор, а компилятор Fortran транслирует программу в исполняемый код. Для объединения подпрограмм, главной программы и набора вспомогательных подпрограмм из стандартных библиотек времени выполнения в единую выполняемую программу используется редактор связей (компоновщик). Завершающим этапом является выполнение программы.

Главной целью разработки начальной версии языка Fortran было обеспечение эффективности выполнения программ. В силу этого языковые структуры были достаточно просты и по большей части имели «корявый вид». В этом смысле можно считать, что Fortran 77 и Fortran 95 стали совсем другими языками. В Fortran 95 уже добавлены почти все современные возможности управления и представления данных, которые отсутствуют в классическом языке, поэтому Fortran 2008 является языком того же уровня, что и языки Ada и C++.

Программа на языке Fortran состоит из главной программы и набора подпрограмм, каждая из которых компилируется отдельно. Окончательное объединение оттранслированных программ в выполняемую форму происходит при загрузке. Каждая подпрограмма компилируется в статически размещаемый сегмент кода и запись активации. Во время выполнения программы уже не происходит никаких изменений в распределении памяти, так как распределение памяти выполняется статически до начала работы программы. Некоторые изменения данной схемы выполнения сделаны для Fortran 95 — в этой версии языка допускается динамическое распределение памяти.

В языке Fortran задано сравнительно мало типов данных: четыре типа числовых данных (целые, вещественные и комплексные числа, а также вещественные с двой-

ной точностью), *булевы* данные, массивы, строки символов и файлы. Поскольку язык предназначен для научных вычислений, в нем имеется большое количество встроенных математических функций и арифметических операций. Также предусмотрены операции отношения, булевы операции и простая выборка элементов массивов при помощи индексов. Поддерживаются файлы последовательного и прямого доступа, присутствует гибкая система ввода-вывода и большой набор возможностей форматирования.

Изначально структуры управления включали: оператор арифметического IF и вызовы функций. Оператор `IF выражение метка1, метка2, метка3` выполнял переход в зависимости от значения вычисляемого выражения; значение могло быть отрицательное, равное нулю или положительное. Управление циклами строилось на основе меток и операторов безусловного перехода `GOTO`. В Fortran 77 добавлены оператор ветвления `IF-THEN-ELSE` и оператор цикла, а в Fortran 90 уже можно было полностью отказаться от операторов `GOTO`. В версии Fortran 90 появилось понятие *устаревшего свойства*, от которого следует отказаться в следующей версии этого языка. Примером может служить оператор арифметического IF.

В языке Fortran предусмотрены два уровня области видимости: глобальная и локальная. Однако в Fortran 90 добавлена концепция *вложенных подпрограмм*. Глобальная область видимости может быть разбита на несколько общих областей (называемых COMMON-блоками; теперь они также находятся в списке устаревших конструкций языка), которые совместно используются несколькими подпрограммами, но только для обращения к элементам данных. Параметры в подпрограммы передаются только по ссылке.

Язык C

Язык C был разработан в 1972 году Дэнисом Ритчи и Кеном Томпсоном из лаборатории *AT&T Bell Telephone Laboratories*. По стилю он похож на Algol и Pascal. Язык C — небольшой, но универсальный язык с ограниченным набором операторов и возможностей. Вместе с тем, органичный сплав простого синтаксиса и высокой эффективности вычислений принесли ему славу самого популярного языка системного программирования [14].

В конце шестидесятых лаборатория *AT&T Bell Telephone Laboratories* закончила совместный с *MIT* и *General Electric* проект по созданию операционной системы Multics, по итогам которого Кену Томпсону захотелось создать более удобную операционную систему. У него зародилась идея операционной системы UNIX, код которой Томпсон решил написать на языке высокого уровня. Он разработал новый язык под названием B, реализующий минимальные возможности для системного программирования. Компилятор B помещался в 8-килобайтную память компьютера. Конечно, сейчас трудно представить, насколько существенным фактором было ограничение емкости памяти всего 40 лет назад.

В 1970 году для проекта UNIX был приобретен компьютер PDP-11 с его огромной памятью в 24 Кбайт. В это время сторонникам UNIX стали мешать ограничения языка B, поэтому были добавлены такие понятия, как типы, определения структур и дополнительные операторы. Новому языку дали новое имя — C.

В семидесятые годы языком C интересовались лишь университеты, использующие операционную систему UNIX. В восьмидесятые годы стали появляться коммерческие версии этой системы. Благодаря языку C популярность UNIX непрерывно росла. В 1989 году увидел свет американский стандарт языка C, который был принят в качестве международного стандарта (ISO/IEC 9899) в 1990 году.

В настоящее время огромное число программистов применяют язык C. Вместе с языками C++ и Java, которые создавались на его основе, он оказывает существенное влияние на стиль программирования. Синтаксис и семантика большинства новых языков (Java, Perl, C#) основаны на понятиях, характерных для языка C.

При практическом программировании на языке C используются:

- *Препроцессор C.* Поскольку каждый компилятор C включает в себя директивы препроцессора, начинающиеся с символа #, то большинство программистов даже не подозревают, что они не являются составной частью самого языка C.
- *Соглашения интерфейса C.* В результате применения языка C возник ряд соглашений. Например, принято, что определения интерфейсов между модулями должны храниться в соответствующем заголовочном файле с расширением `.h`. В директиве `#include "myappl.h"` при определении интерфейса к модулю `myappl` используется как препроцессор C, так и указанное соглашение.
- *Библиотека C.* Многие функции (`printf`, `getchar`, `malloc`, `fork` и `exec`) были написаны с помощью интерфейсов C, хотя они и не входят в официальное описание языка C. Однако в стандарте языка C эти функции уже включены в язык (как обязательные библиотечные функции для стандартных компиляторов). Подключение большой библиотеки позволяет существенно расширять функциональные возможности ядра языка.

Модуль на языке C состоит из глобальных объявлений и последовательности вызовов функций. Для формирования одной выполняемой программы одновременно загружается несколько модулей. Каждая функция может вызывать другие функции и имеет доступ как к локальным, так и к глобальным переменным.

Структура хранения данных достаточно проста. Каждая функция имеет локальную память или локальную запись активации, которая является динамической и допускает рекурсию. Кроме того, каждая функция имеет доступ к глобальным переменным.

В языке C применяются указатели, причем понятия массива и указателя тесно связаны, что позволяет программам легко обращаться к элементам массивов. Строки рассматриваются как массивы символов.

Язык C содержит большой набор арифметических операций, которые позволяют писать очень эффективные и лаконичные программы. Предусмотрен полный набор операторов управления с весьма гибкой семантикой, допускающей довольно необычное использование.

Предусмотрена возможность гибкого определения типов. Правда, C считается языком со слабой системой типизации. Дело в том, что большинство отдельных типов данных считаются подтипами целого типа. Хотя транслятор и обнаружит ошибки, связанные с несоответствием типов, но поскольку большинство элементов данных относятся к целочисленному типу, многие ошибки могут остаться незамеченными.

Изначально язык C был сильно интегрирован с операционной системой. В операционной системе UNIX некоторые функции (например, функция `malloc` для динамического выделения области памяти) заданы как вызовы отдельных функций языка C. По соглашению все они определяются в системных заголовочных файлах с расширением `.h`. В частности, для обеспечения вызова `malloc` в начале программы на языке C должна быть записана директива `#include <malloc.h>` подключения соответствующего файла, а при дальнейшем использовании этой функции в программе следует просто писать `malloc(Size)`, указывая в качестве параметра требуемую емкость памяти. При компиляции и запуске такой программы определения соответствующих функций (например, `malloc`) подключаются из библиотеки C.

Компилятор C прежде всего запускает препроцессор. Сначала выполняются такие директивы, как `#define` и `#include`, а затем уже транслятором C компилируется вся остальная программа.

Большинство функций ввода-вывода определено в системном файле `stdio.h`, который должен подключаться к любой программе на языке C. Такой подход позволяет легко расширять язык: достаточно лишь написать ряд новых функций (для добавления новых функциональных возможностей).

В программе можно использовать комментарии (любой текст, ограниченный символами /*...*/), которые задаются в любом месте, где можно использовать пробел.

Функциональные языки программирования

Аппликативный подход к вычислениям, в отличие от императивного подхода, основывается на собственной идее. Вместо отслеживания последовательности состояний, через которые должен пройти компьютер для получения ответа, здесь ищется функция программы, обеспечивающая ожидаемый результат.

Внимание концентрируется на последовательных преобразованиях начальных данных с помощью функций. Как только будет придуман конечный ответ (последовательность функций), его можно применить к начальным данным и получить результат.

Функциональные (аппликативные) языки задают вычисления как вызовы функций.

Синтаксис функциональной программы выглядит следующим образом:

$function_n(\ldots function_2(function_1(данные))\ldots)$

В начале вычислений к исходным данным применяется функция $function_1$, затем к полученному промежуточному результату — $function_2$ и т. д. В последнюю очередь применяется функция $function_n$. Она возвращает ожидаемый результат.

Таким образом, в функциональных языках отсутствуют переменные (в том смысле, который им придается в императивных языках), нет операторов присваивания, нет повторений в форме итераций.

Чтобы определить функцию, нужно придумать ее имя и задать вычисляемое ей выражение. Описание единичного действия — это вызов функции с конкретными аргументами. Описание программы — это описание последовательности вызовов отдельных функций.

Рассмотрим пример. Положим, что надо вычислить наибольшее из шести чисел a, b, c, d, e, f.

Будем считать, что в функциональном языке разрешена функция только для двух аргументов:

$$MAX(x, y) = \begin{cases} x, \text{ если } x >= y, \\ y, \text{ если } x < y. \end{cases}$$

В этом случае потребуется группировать числа попарно и обеспечить многократные вызовы функции *MAX* (*x*, *y*).

В качестве первого приближения запишем:

$$MAX\ (a, b, c, d, e, f) = MAX\ (MAX\ (a, b), MAX4\ (c, d, e, f)), \quad (3.1)$$

где псевдофункцию для четырех аргументов определим в виде:

$$MAX4\ (c, d, e, f) = MAX\ (MAX\ (c, d), MAX\ (e, f)). \quad (3.2)$$

Подставив (3.2) в (3.1), получим окончательную форму функциональной программы:

$$MAX\ (MAX\ (a, b), MAX\ (MAX\ (c, d), MAX\ (e, f))).$$

Ее выполнение дает искомый результат.

Еще один пример. Положим, что в функциональном языке поддерживается лишь одна функция:

$$MIN(x, y) = \begin{cases} x, \text{ если } x <= y, \\ y, \text{ если } x > y. \end{cases}$$

В этом случае для вычисления наименьшего из шести чисел a, b, c, d, e и f следует использовать программу:

$$MIN\ (MIN\ (a, b), MIN\ (MIN\ (c, d), MIN\ (e, f))).$$

Обсудим преимущества и недостатки функционального программирования.

Прежде всего, функциональные программы предлагают предельно естественную, математическую форму записи вычислений. Программы получаются короткими, выразительными и понятными. Здесь доминирует более высокий уровень программирования, требующий меньших усилий, чем императивное программирование. Программист свободен от необходимости учитывать те детали, которыми наполнена императивная программа.

Однако за высокий уровень абстракции приходится расплачиваться проблемами реализации. В функциональном языке программисту не нужно связываться с переменными, поскольку здесь не требуются такие абстракции, как ячейки памяти. Для организации повторения действий используется мощный механизм рекурсии, тоже оставляющий в стороне детали реализации. Все это выливается в большие накладные расходы времени и памяти при выполнении программы. Компьютер с последовательным порядком работы и линейной организацией памяти просто не способен обеспечить высокую эффективность функциональных вычислений.

Перечислим наиболее известные функциональные языки. Первый язык функционального программирования LISP появился в рамках проекта по искусственному интеллекту (1960). Язык Scheme (1975) был построен как диалект языка LISP. Это небольшой язык с простым синтаксисом и семантикой, удобный для обучения студентов.

На протяжении 1970-х и в начале 1980-х годов разрабатывалось множество различных диалектов языка LISP, в деталях отличавшихся друг от друга. Для устранения этой ситуации была воссоздана стандартная версия языка — COMMON LISP (1984).

В начале 1980-х Дэвид Тернер из университета Кента в Кентербери предложил английский вариант функционального языка с названием Miranda (1986).

Язык ML (MetaLanguage) был создан Робином Милнером в Эдинбургском университете (1990). ML поддерживал не только функциональную, но и императивную парадигму. В этот язык вернулись понятия переменной, типа и оператора. Синтаксис языка ML подобен синтаксису императивных языков.

Язык Haskell (1992) как чисто функциональный язык считается современным преемником языка Miranda. Интересна использованная здесь идея «ленивого» вычисления: никакое выражение не вычисляется до тех пор, пока не потребуется его значение.

Язык LISP

Язык LISP был разработан группой исследователей под руководством Джона Маккарти из Массачусетского технологического института (МТИ) в 1960 году. Этот язык широко использовался для научных исследований в области естественных языков, доказательства теорем, задач искусственного интеллекта. За полвека создано большое количество версий языка LISP [64, 75].

Наиболее удивительная черта LISP — это эквивалентность форм представления в языке программ и данных, что позволяет выполнять структуры данных как программы и изменять программы как данные. Другая характерная особенность: в качестве структуры управления применяется рекурсия, а не итерация (цикл), как в императивных языках. В большинстве программ на LISP широко используются рекурсивные вызовы функций.

С самого начала LISP подвергался критике за медленность вычислений на стандартном компьютере фон Неймана.

Программы на LISP выполняются в диалоговой среде. Поэтому не существует главной программы в традиционном понимании. Вместо этого пользователь вводит последовательность выражений, которые требуется вычислить. Система LISP вычисляет выражения по мере их ввода и автоматически выводит результаты на экран монитора. Некоторые введенные выражения являются определениями функций. Другие выражения содержат обращения к этим функциям с указанием конкретных значений аргументов. В LISP не существует блочной структуры или других сложных структур. Единственной формой взаимодействия между функциями считается обращение к одной функции из другой во время выполнения программы.

Функции LISP всегда определяются как выражения. Каждый оператор является функцией, возвращающей некоторое значение, а подпрограммы записываются как

единые (иногда весьма сложные) выражения. Базовый синтаксис LISP определяет только выражения, но для увеличения сходства с обычным синтаксисом программы в язык были добавлены разнообразные специальные конструкции.

Набор типов данных в LISP довольно мал. Основными элементарными типами данных являются *литеральные атомы* (символы) и *числовые атомы* (числа). Связанные списки и списки свойств (особая разновидность связанных списков) образуют основные структуры данных.

LISP имеет широкий набор элементарных операций для создания, уничтожения и модификации списков. Дополнительно поддерживаются основные арифметические операции. Во время выполнения программы с помощью специальных элементарных операций можно транслировать и выполнять другие программы, а программы могут создаваться и выполняться динамически. Предусмотрено распараллеливание действий: во время выполнения одной программы можно транслировать и выполнять другие программы

Структуры управления LISP очень просты. Используемые выражения записываются в кембриджской польской записи и могут содержать условное ветвление. Конструкция `prog` задает простую структуру для записи последовательности выражений.

Комментарии в LISP начинаются точкой с запятой, при этом вся следующая за этим символом строка является комментарием.

Обработка нелокальных ссылок в LISP основана на правиле последней ассоциации, которое реализуется с помощью простого связанного списка текущих ассоциаций (он называется *A-списком*). При встрече ссылки на какой-либо идентификатор в этом списке для него отыскивается текущая ассоциация.

Параметры функций передаются по значению или по имени, в зависимости от классификации функции. Обычным случаем считается передача параметров по значению.

Проще всего LISP реализуется с помощью программного интерпретатора и программного моделирования всех элементарных операций. Дополнительно предоставляется компилятор, который можно использовать для компиляции определений функций в машинные коды. Эти скомпилированные функции затем выполняются аппаратным интерпретатором (но по-прежнему для многих операций требуется программное моделирование). LISP довольно плохо приспособлен для компиляции, поскольку большая часть связываний происходит только во время выполнения программы. В качестве основного средства для хранения данных и программ применяется сложная система управления памятью (на основе кучи и сбора мусора).

Логические языки программирования

Логические (декларативные) языки описывают не способ решения задачи, а саму задачу. Фактически они задают формализованное представление предметной области.

Иногда логические языки называют *языками, основанными на системе правил*. Здесь осуществляется проверка некоторого разрешающего высказывания (утверждения), и в случае его истинного значения выполняется соответствующее действие.

В императивном языке программирования вычисления описываются очень подробно, с обязательным указанием порядка выполнения операторов. В языке, основанном на системе правил, не существует определенного порядка применения правил, и система реализации языка должна сама выбрать нужный порядок выполнения действий, который приведет к желаемому результату. Такая парадигма программирования радикально отличается от парадигм, используемых в остальных трех видах языков программирования, и, безусловно, требует абсолютно иной языковой поддержки.

Синтаксис программ для логических языков выглядит следующим образом:

```
действие1 :- разрешающее высказывание1
действие2 :- разрешающее высказывание2
…
действиеn :- разрешающее высказываниеn
```

Подчеркнем еще раз: действия осуществляются не в той последовательности, в которой они определены в программе. Порядок выполнения определяют разрешающие высказывания, которые, по сути, являются предикатами первого порядка. Более подробно об этом будет рассказано в главе 21.

Пусть существует некоторая логическая машина.

Она имеет:

- набор фактов

```
любит (том, футбол)
любит (боб, баскетбол)
любит (ник, теннис)
```

- аксиомы (правила вывода)

```
любит (том, X) : – любит (билл, X)
```

В этом описании приняты следующие обозначения:

1. Утверждения типа «Том любит футбол» представляются в виде:

```
любит (том, футбол)
```

2. Утверждение типа «Если Билл любит что-то, то это же любит Том» записывается как аксиома:

```
любит (том, X) : – любит (билл, X)
```

Здесь правая часть — это исходное условие (высказывание), а левая часть — заключение.

3. Имена постоянных пишутся с маленькой буквы (том), а имена переменных — с большой (X).

Программа — это запрос к логической машине

```
? любит (билл, Y)
```

означающий «Что любит Билл?»

Цель выполнения программы: замена переменной на постоянную. Для этого просматриваются факты и аксиомы, ищутся возможные *подстановки*. Они могут выполняться неоднократно.

Подстановки для нашей программы:

```
1. любит (том, Y)
2. любит (том, футбол)
```

В результате вместо Y подставляется футбол.

Примеры других программ, применимых к нашей логической машине:

```
? любит (Y, футбол)
? любит (боб, Z)
? любит (том, футбол)
```

Последняя программа — это требование подтверждения факта.

Самый известный язык логического программирования получил название Prolog.

Он создан авторским коллективом из преподавателей Марсельского и Эдинбургского университетов (Франция, 1972).

В 1981 году Япония инициировала государственный проект по созданию компьютеров нового поколения, рабочим языком которых должен был стать Prolog. Увы, проект провалился (из-за ограниченных возможностей существующей элементной базы).

Язык Prolog

Язык Prolog ориентирован на решение задач с использованием исчисления предикатов [4, 15, 23].

Язык Prolog предоставляет возможность задания спецификаций решения и позволяет компьютеру вывести из них последовательность выполнения для этого решения. Он не задает алгоритм решения задачи. Например, если информация об авиарейсах представлена в следующей форме:

```
flight(номер_рейса, пункт_отправления, пункт_назначения, время_отправления, время_
прибытия)
```

тогда все рейсы из Риги в Даугавпилс можно задать либо в виде прямых рейсов с помощью конструкции

```
flight(номер_рейса, Рига, Даугавпилс, время_отправления, время_прибытия)
```

либо в виде рейсов с промежуточной посадкой

```
flight (номер_рейса1, Рига, X, время_отправления1, время_прибытия1),
flight (номер_рейса2, X, Даугавпилс, время_отправления2, время_прибытия2),
время_отправления2 >= время_прибытия1+30
```

Это означает, что вы определяете город X, в который можно попасть рейсом из Риги и откуда можно улететь в Даугавпилс, и самолет в Даугавпилс вылетает через 30 минут после прилета рейса из Риги, чтобы осталось время на пересадку. Здесь не задан никакой алгоритм, записаны лишь условия для получения правильного решения. Если мы сможем указать подобный набор условий, среда поддержки языка сформирует последовательность действий, необходимую для выбора подходящего рейса.

Разработка языка Prolog началась в 1970 году Аланом Кулмероэ и Филиппом Русселом. Они намеревались создать язык, который мог бы обеспечивать логические выводы на основе заданного текста. Название Prolog было выбрано как сокращение от «Programming in logic». Этот язык был создан в Марселе (Франция) в 1972 году. За основу был взят принцип резолюции Роберта Ковальски, сотрудника Эдинбургского университета. Принцип резолюции позволил разработать механизм логического вывода. При применении формул Хорна была обеспечена

эффективная унификация, в которой недетерминизм устранялся за счет процесса отката. Алгоритм резолюции обеспечивал генерацию последовательности действий, реализующей спецификации, подобные `flight`.

Программа на языке Prolog состоит из набора фактов, а также отношений, которые устанавливаются между объектами данных (фактами) и набором правил (образцами отношений между объектами базы данных). Факты и правила заносятся в базу данных с помощью операции `consult`. Чтобы программа начала работу, пользователь должен ввести запрос. Запрос образует набор термов, каждый из которых должен иметь истинное значение. Факты и правила из базы данных определяют такие подстановки для переменных в запросе (они называются *унификацией*), которые согласуются с информацией в базе данных.

Система реализации языка Prolog приглашает пользователя вводить информацию. Пользователь набирает запрос или имя функции. Выводится значение этого запроса (*истина* или *ложь*), а также возможные значения переменных запроса, присвоение которых делает запрос истинным, иначе говоря, *унифицирует* запрос. Если ввести символ «;», тогда отображается следующий набор значений переменных, унифицирующий запрос, и так до тех пор, пока не исчерпаются все варианты возможных подстановок, после чего система выводит `no` и ждет следующего запроса. Нажатие `Enter` воспринимается как указание «прекратить поиск дополнительных решений».

Вычисления в системе языка Prolog основываются на спецификациях предикатов, но схожи с выполнением функциональных программ на языках LISP или ML. При создании правил для языка Prolog используется то же рекурсивное мышление, что и при разработке функциональных программ.

В языке Prolog применяется простой синтаксис и простая семантика. Поскольку здесь ищутся отношения между объектами, основными формами представления данных являются переменная и список. Правило подобно процедуре, хотя механизм унификации более сложен, чем механизм подстановки параметров в выражения.

Объектно-ориентированные языки программирования

Парадигма объектно-ориентированного программирования является развитием императивного программирования. При ее создании преследовались две цели:

- сократить размеры программ за счет повышения размера строительных элементов («маленькие» переменные заменяются «большими» объектами) и тем самым обеспечить возможность создания (за то же время) более крупных программных приложений;
- упростить процесс создания новых программ на базе старых (за счет применения механизма наследования).

Объектно-ориентированные языки впитали в себя лучшие понятия и механизмы императивных языков. **Объектно-ориентированные языки задают вычисления как взаимодействия программных объектов.**

Объект — это именуемый модуль, заключающий в себе данные и операции для их обработки.

Программный объект во многом похож на физический объект реального мира. В частности, программный объект имеет свое состояние и демонстрирует окружающей среде свое поведение.

Состояние объекта характеризуется перечнем данных и их значений. Поведение задается последовательностью выполняемых операций.

Объекты взаимодействуют друг с другом с помощью сообщений. Посылается сообщение объектом-источником в адрес объекта-приемника. Каждое сообщение — это запрос на выполнение операции объектом-приемником.

Описание объектов с общей структурой и поведением называется *классом*. Как и переменные, единичные объекты создаются по их описаниям. Только в роли описаний для переменных выступают типы данных, а в роли описаний для объектов — классы. Объект считается экземпляром класса.

Специфику объектно-ориентированного подхода определяют три принципа:

- инкапсуляция (сокрытие своего содержимого от внешнего мира);
- наследование (возможность получения потомками структуры и поведения предков);
- полиморфизм (использование одного и того же имени для выражения различных действий и объектов).

Инкапсуляция означает, что каждый объект помещен в защитную оболочку, сквозь которую другие объекты видят лишь самое необходимое: заголовки операций, которые может выполнять объект. Наследование позволяет внедрить в новый класс элементы данных и операции старого класса, обеспечивая возможность их модификации. Наконец, полиморфизм поддерживает возможность существования целого семейства различных операций с одинаковым именем.

Первый объектно-ориентированный ЯП, Simula 67, был придуман норвежцами К. Нигаардом и У. Далом как расширение императивного языка Algol 60. Этот язык опередил свое время и был благополучно забыт.

Второй объектно-ориентированный ЯП, Smalltalk, появился в нужное время (1972–1980) и в нужном месте. Его автором стал Алан Кей из фирмы *Xerox*.

Широкое распространение получил язык C++, созданный Б. Страуструпом (1983).

Заметим, что очень часто используются гибридные языки, реализующие сразу несколько парадигм: Ada 2005, Eiffel, Object Pascal (императивная и объектно-ориентированная парадигмы), CLOS (функциональная и объектно-ориентированная парадигмы).

Язык Smalltalk

Язык Smalltalk создавался в рамках целой системы, а не как отдельный инструмент для написания программ. Объектная ориентация изначально встроена в этот язык, в отличие от других объектно-ориентированных языков, где к уже существующим императивным механизмам была просто добавлена возможность наследования [63].

Smalltalk был придуман в начале семидесятых годов Аланом Кеем из исследовательского центра *Xerox PARC*. Целью работы было создание целой среды для

персонального компьютера. В 1972 году Дан Ингаллс разработал первую реализацию языка под именем Smalltalk-72. После некоторых модификаций общепринятым стандартом стал Smalltalk-80.

Первоначально программы на Smalltalk выполнялись только в специализированной вычислительной среде. В настоящее время имеются реализации этой среды, которые легко устанавливаются на обычные компьютеры.

Уникальность языка Smalltalk определяют следующие характеристики.

Среда разработки. Smalltalk создавался как целая система, элементами которой являются язык, аппаратные средства компьютера, операционное окружение и среда программирования.

Минимальный язык разработки. В самом языке Smalltalk описан лишь небольшой набор базовых средств. С их помощью можно создать класс (как подкласс других классов) и прикрепить к нему методы. Основные операторы Smalltalk задают присваивание и последовательность действий. Истинная мощь языка обеспечивается параметрическим механизмом сообщений. При помощи сообщений можно определять структуры ветвления и циклы.

В исходной поставке системы Smalltalk прописан набор встроенных классов. В ходе инсталляции на компьютере формируется образ системы с этими классами и пустым окружением. При вызове интерпретатора Smalltalk программист получает собственную копию этого образа, внутри которой создает новые классы и модифицирует существующие классы (добавлением или изменением методов). Системный администратор может изменить систему Smalltalk, предложив другой набор предопределенных классов.

Выполнение программы Smalltalk. Работа Smalltalk основана на модели взаимодействия. Данные в Smalltalk образуются объектами, а методы рассматриваются как сообщения, посылаемые объектам. Например, 3 + 2 означает, что сообщение «+» с параметром 2 послано целочисленному объекту 3. Метод «+» в данном примере возвращает значение 5. Конечно, к такой нотации надо привыкнуть.

Smalltalk применяет динамическую модель последовательности действий. Каждый метод выполняется с одновременным созданием записи активации. Поскольку блоки, которые содержат локальные переменные, могут быть присвоены переменной, стековая организация памяти с дисциплиной LIFO здесь не применима. Для хранения объектов данных (и записей активации) Smalltalk реализует динамическую организацию памяти в виде кучи. Для очистки памяти выполняется динамическая сборка мусора.

Язык разметки HTML

Язык HTML (HyperText Markup Language — язык разметки гипертекста) предназначен для создания гипертекстовых документов в среде WWW (World Wide Web — Всемирная паутина). HTML-документы (иначе веб-документы, или веб-страницы) могут просматриваться различными типами веб-браузеров. HTML — один из наиболее простых языков создания веб-страниц [21, 40].

У каждого компьютера в Интернете имеется свой адрес. Когда на веб-сервер приходит запрос, адрес отправителя всегда известен. Ответ посылается именно

по этому адресу. Обеспечивают взаимодействие сервера и клиента несколько протоколов передачи данных.

Первый из протоколов — это TCP/IP (Transmission Control Protocol/Internet Protocol). После установления соединения по TCP/IP запускается веб-браузер, использующий протокол передачи гипертекста (http, hyper text transfer protocol) для организации обмена данными. Веб-сервер посылает HTML-документы на браузер клиента, который выводит их на экран.

У каждой веб-страницы есть свой адрес. Эти адреса называются унифицированными указателями информационных ресурсов URL (Uniform Resource Locators). Например, URL для некоторого веб-документа на сайте корпорации Microsoft имеет вид:

```
http://www.microsoft.com/windows/index.html
```

В приведенной записи `http://` именует протокол, `www.microsoft.com` — это адрес веб-сервера компании Microsoft. Далее следует слеш — косая черта (`/`), обозначающая, что после нее следует путь. Путь к файлу указывает на документ `index.html`, находящийся в папке `windows`.

Описания документов в языке HTML создаются с помощью элементов. *Элемент* — это команда, записываемая тегами. Говорят, что элементы задают разметку текста документа. В общем случае элемент имеет следующую структуру:

```
Открывающий_тег Содержание Закрывающий_тег
```

Содержание — это некоторая последовательность символов. Все теги начинаются с левой угловой скобки (`<`) и заканчиваются правой угловой скобкой (`>`). Внутри угловых скобок указывается имя тега и, в случае открывающего тега, атрибуты. Имя открывающего тега формирует имя всего элемента. Имя открывающего тега повторяется в закрывающем теге, но предваряется символом косой черты (`/`). Например, элемент, определяющий заголовок документа, записывается следующим образом:

```
<title> Заголовок документа </title>
```

Здесь тег `<title>` говорит веб-браузеру о вхождении в документ элемента `заголовок`, а тег `</title>` — о завершении текста заголовка.

Достаточно часто теги элемента называют контейнером, который заполняется конкретным содержанием.

Атрибуты — это имена свойств тегов (элементов). Они отделяются друг от друга пробелами. Атрибуты могут принимать определенные значения. Все значения атрибутов должны быть заключены в двойные кавычки. Атрибуты имеются только у открывающих тегов, у закрывающих тегов их нет. Атрибуты влияют на результат интерпретации элемента браузером.

Например, в открывающем теге `<body>`

```
<body bgcolor="#ffff05" topmargin="0" bottommargin="0" leftmargin="0" rightmargin="0">
```

с помощью атрибутов заданы: цвет фона документа (атрибут `bgcolor`), верхняя граница страницы в пикселах (атрибут `topmargin`), нижняя граница страницы (атрибут `bottommargin`), левая граница страницы (атрибут `leftmargin`) и правая граница страницы (атрибут `rightmargin`).

Для формирования структуры документа используют элементы `html`, `head`, `title`, `body`. Когда веб-браузер получает документ, он по элементам определяет, как документ должен быть интерпретирован.

Корневой элемент *html*. Самый первый элемент, который встречается в документе, должен начинаться с тега `<html>`. Данный тег сообщает веб-браузеру, что документ написан на языке HTML.

Структурно документ делится на две части: заголовочную и основную, или тело документа. Заголовочная часть размещается между тегами `<head>` и `</head>`, основная — между тегами `<body>` и `</body>`.

Элемент *head*. Между тегами `<head>` и `</head>` могут размещаться элементы `title` (текст заголовка), `meta` (служебная информация), `style` (стили на странице), `script` (добавление сценария), `link` (ссылка на внешние стили), `base` (базовый адрес документа), `basefont` (шрифт на странице) и `bgsound` (фоновый звук на странице). Данный элемент реализует исключительно структурирующие функции, его теги не имеют никаких атрибутов.

Элемент *title*. Внутри тегов `<head>` и `</head>` размещается единственный обязательный элемент из тегов `<title>` и `</title>`, содержащий текст заголовка.

Элемент заголовочной части документа `head` должен быть указан сразу после элемента `html` и более нигде в теле документа. Открывающий тег `<head>` указывается непосредственно перед тегом `<title>`, а закрывающий тег `</head>` — сразу после окончания описания документа. Заголовок документа при отображении самого документа в окне браузера не виден.

Элемент *body*. Элемент `body` предназначается для выделения той части документа, которая должна быть показана пользователю на экране. Тело документа должно находиться между тегами `<body>` и `</body>`. Тег `<body>` имеет достаточно большое количество атрибутов. В этой части документа содержится вся текстовая и графическая информация, размещаются все элементы, которые образуют основное содержание документа. Порядок следования элементов здесь именно такой, в каком они отображаются на экране.

Минимальный документ HTML имеет следующий синтаксис:

```
<html>
    <head>
        <title> Заглавие документа </title>
    </head>
    <body> Текст документа </body>
</html>
```

Текст тела HTML-документа может иметь сложную структуру, разметка которой обеспечивается многочисленными элементами языка HTML. Опишем лишь некоторые из предоставляемых возможностей.

Разделы. Они разделяют части документа. Элемент `<h1>...</h1>` — это заголовок первого уровня. Он печатается крупными буквами. Теги `<h2>` и `<h3>` используются для задания подзаголовков — заголовков второго и третьего уровней, которые отображаются шрифтами меньшего размера.

Списки. Списки элементов могут отображаться различными способами: тег `<li>` начинает следующий элемент списка, который завершается либо концом самого списка — `</li>`, либо началом следующего элемента списка (тогда закрывающий тег отсутствует). Возможны нумерованные списки, задаваемые тегом `<ol>` (ordered list — упорядоченный список), или неупорядоченные списки, задаваемыми тегом `<ul>` (unordered list — неупорядоченный список).

Таблицы. Таблицы создаются с помощью элемента `<table>...</table>`. Таблицы могут состоять из произвольного количества строк и столбцов. Каждая строка в таблице отмечается элементом `<tr>...</tr>`, а ячейка — элементом `<td>...</td>`. Заголовок таблицы образует элемент `<th> заголовок </th>`.

Гиперссылки. Гиперссылки являются ключевым средством, делающим веб-страницы столь привлекательными для пользователей. Именно гиперссылки стали главной причиной ошеломляющей популярности Всемирной паутины, в которой пользователь может легко переходить от одной страницы к другой простым щелчком мыши. HTML позволяет организовать связь с другим документом HTML, имеющим известный URL-адрес. Ссылка на другой документ HTML задается тегом `<a>`, включающим в себя атрибут `href`. Например, элемент `<a href="url-адрес"> описание </a>` позволит веб-браузеру выделить строку `описание`. Щелчок по выделенной строке обеспечит обращение браузера к тому документу HTML, который расположен в сети по указанному URL-адресу. Этот документ отобразится в окне браузера. Данный механизм лежит в основе всех путешествий по сети.

Скриптовые языки

Традиционные языки программирования предназначены прежде всего для создания автономных приложений: программ, которые принимают входные данные, обрабатывают их определенным образом и генерируют соответствующие результаты. Однако в большинстве компьютерных приложений требуется координация множества программ. В большой платежной системе, например, должны обрабатываться данные от устройств чтения кредитных карточек, сканируемых бумажных форм а также данные, вводимые вручную с клавиатуры. Здесь выполняются тысячи запросов к базе данных; реализуются сотни юридических и корпоративных правил; создается обширная документация для сохраняемых записей, аудита, налоговых отчислений; распечатываются ведомости по зарплате; поддерживается связь с серверами по всему миру для обеспечения депозитов в режиме «онлайн», налогового обслуживания, изъятия накоплений, медицинского страхования и т. д. Эти задачи, вероятно, требуют применения сотен отдельных программ. Координация этих программ бесспорно потребует разнообразных проверок и логических решений, циклов, переменных и типов, подпрограмм и абстракций — тех же видов логических утилит, которые традиционные языки предоставляют внутри приложения.

В гораздо меньших масштабах художник-график или фотокорреспондент загружает изображения из цифровой камеры, конвертирует их в любимый формат, вращает изображения для выбора оптимальной ориентации, индексирует их по времени, теме и цветовой гамме, перемещает в удаленный архив, а затем повторно инициализирует память камеры. Ручное выполнение этих шагов, вероятно, будет и утомительно, и подвержено ошибкам. В подобном стиле создание динамической веб-страницы может требовать идентификации и авторизации, поиска в базе данных, отображения манипуляций, удаленного взаимодействия, чтения и записи текста HTML. Сценарии подобных действий принято называть *скриптами*. Все эти сценарии обосновывают потребность в программах, которые координируют другие программы.

Конечно, можно написать программный код координации на Java, C или некотором другом традиционном языке, но это не всегда просто. Традиционные языки повышают эффективность, сопровождаемость, мобильность и возможность статического обнаружения ошибок. Их системы типизации создаются на основе таких понятий аппаратного уровня, как целые числа фиксированного размера, числа с плавающей запятой, символы и массивы. Напротив, *скриптовые языки* ориентированы на повышение гибкости, скорости разработки, локализацию и динамическую проверку (в период выполнения). Их системы типизации включают в себя такие понятия высокого уровня, как таблицы, шаблоны, списки и файлы.

Универсальные скриптовые языки, такие как Perl и Python, иногда называют *склеивающими языками*, поскольку они создавались для «склеивания» существующих программ в интересах создания большой системы. С ростом Всемирной паутины скриптовые языки получили новый толчок в направлении генерации динамического информационного наполнения сети. Они также широко используются как *языки расширения*, которые позволяют пользователю настраивать или расширять функциональные возможности «скриптовых» утилит.

Современные скриптовые языки происходят от предков двух категорий. В одной категории находятся командные интерпретаторы или «оболочки» традиционных пакетных или «терминальных» вычислений (в режиме командной строки). В другой категории — различные утилиты для обработки текста и генерации отчетов.

Примеры из первой категории предков включают в себя язык управления заданиями *IBM*, интерпретатор команд операционной системы MS-DOS, оболочки SH (от слова оболочка — shell) и CSH семейства Unix. Ранние языки данной категории предлагали занести список команд, называемый скриптом (сценарием), в файл, который нужно интерпретировать. Язык SH начинался с небольшой коллекции команд, которые интерпретировались как вызовы системных подпрограмм, выполняющих вспомогательные функции (управление файлами, простая фильтрация файлов). К коллекции были добавлены переменные, операторы управления порядком вычислений, функции и другие возможности; в результате появился полный язык программирования. Одним из самых мощных и широко известных в этой категории был язык KSH (Bolsky и Korn, 1995), разработанный Дэвидом Корном в *Bell Laboratories*.

Примерами из второй категории предков являются генератор отчетов *IBM*, а также SED (потоковый редактор — stream editor) и язык AWK для Unix (назван в честь своих авторов — Альфреда Ахо (Alfred Aho), Питера Вейнбергера (Peter Weinberger) и Брайана Кернигана (Brian Kernighan)). AWK стартовал как язык генерации отчетов, но позднее превратился в язык более широкого назначения. От них развился язык Perl, первоначально разработанный Ларри Уоллом (Larry Wall) в 1986 году, а теперь ставший наиболее широко используемым универсальным скриптовым языком. Другими универсальными скриптовыми языками являются Tcl (Tickle), Python, Ruby, VBScript (для Windows) и AppleScript (для Mac).

С ростом Всемирной паутины в конце 90-х язык Perl был сильно адаптирован для создания веб-скриптов серверной стороны, на которой веб-сервер выполняет программу (на машине сервера) для генерации страницы. Одним из энтузиастов веб-скриптов был Расмус Лердорф (Rasmus Lerdorf), который создал коллекцию сценариев для трассировки обращений к его персональной домашней странице.

Первоначально написанные на языке Perl, но вскоре переписанные в виде законченного и независимого языка, эти сценарии развились в PHP, очень популярный теперь инструмент для описания скриптов веб-серверной стороны. Конкурентами PHP являются JSP (Серверные Страницы Java) и VBScript (на платформе Microsoft). Для создания скриптов на клиентском компьютере все основные браузеры реализуют язык JavaScript, разработанный корпорацией Netscape в середине 90-х и стандартизированный Европейской ассоциацией производителей компьютеров в 1999 году.

В своей классической работе по созданию скриптов [77] Джон Аустерхут (John Ousterhout) отмечает: «В скриптовых языках предполагается, что уже существует коллекция полезных компонентов, написанных на других языках. Скриптовые языки предназначены не для того, чтобы писать приложения с нуля, а скорее для того, чтобы объединять компоненты». Дж. Аустерхут прогнозирует будущее, в котором программисты активно применяют скриптовые языки для создания высокоуровневых системных структур, где крайне важны ясность, повторное использование и простота разработки. Традиционные языки, такие как C, C++ или Java, считает он, будут использоваться для модульных, повторно используемых системных компонентов, где важна сложность алгоритмов или скорость выполнения. Общее эмпирическое правило, по его словам, имеет вид: программный код на языке скриптов может быть разработан в 5–10 раз быстрее, но выполняться будет в 10–20 раз быстрее, если написан на традиционном языке программирования.

Некоторые авторы резервируют термин «создание скрипта» для склеивающих языков, используемых для координации множества программ. В общем случае, однако, создание скрипта (сценария) — более широкое и более неопределенное понятие. Нет никаких сомнений, что оно включает в себя создание веб-скриптов. Кроме того, большинство авторов распространяет его и на языки расширения.

Одни пользователи знакомы с «макросами» на Visual Basic для Microsoft Office. Другим пользователям известен текстовый редактор Emacs, написанный на языке, являющемся расширением языка LISP. Несколько языков (Tcl, Python, Guile и Elk), созданных как диалекты языка Scheme, имеют реализации, ориентированные на расширение больших программных приложений. Расширение считается основной целью языка Tcl. Некоторые широко используемые коммерческие приложения предоставляют свои собственные языки расширения. Например, инструментарий Tk, первоначально созданный для использования с Tcl, был включен в несколько скриптовых языков (Perl, Python, Ruby) для программирования графического интерфейса пользователя GUI.

В качестве языка скриптов можно рассматривать и расширяемый язык стилей для преобразований XSLT. XSLT — это часть растущего семейства утилит для расширенного языка разметки XML.

Общие характеристики скриптовых языков

Скриптовые языки — это быстрорастущая популяция языков программирования. Довольно трудно дать точное определение скриптовых языков, однако у них есть несколько общих характеристик.

Пакетное и интерактивное использование. В некоторых скриптовых языках (например, в языке Perl) используется компиляция «на лету», при которой для

генерации любого результата читается весь исходный текст программы. Большинство других языков ориентировано на построчную компиляцию или интерпретацию. Например, Python, Tcl, Guile и Ruby (с коротким вспомогательным скриптом) воспринимают все команды с клавиатуры.

Экономичность выражений. Для поддержки быстрой разработки и интерактивного использования скриптовые языки ориентированы на применение минимума «шаблонов». Некоторые интенсивно используют пунктуацию и очень короткие идентификаторы (язык Perl), в то время как другие (Tcl, AppleScript) ориентированы на синтаксис естественной речи, с большим количеством слов и небольшим количеством пунктуации. Все скриптовые языки пытаются избежать развернутых объявлений и высокоуровневой структуры, обычных для традиционных языков.

Отсутствие объявлений; простые правила видимости данных. Большинство скриптовых языков обходится без объявления, в них предоставляются простые правила видимости имен. В некоторых языках (например, в языке Perl) все конструкции являются глобальными по умолчанию; редкие объявления задают для переменной вложенную область видимости. В других языках (например, PHP и Tcl) все данные считаются локальными по умолчанию; глобальные переменные должны быть явно импортированы. В языке Python принято правило, по которому любая присваиваемая переменная локальна в блоке, в котором появилось присваивание. Для присваивания переменной в контексте внешней области видимости требуется специальный синтаксис.

Гибкая динамическая типизация. В условиях отсутствия объявлений большинство скриптовых языков типизировано динамически. В некоторых (например, PHP, Python, Ruby и Scheme) тип переменной проверяется непосредственно перед использованием. В других (например, Perl и Tcl) переменная будет интерпретироваться по-разному в различных контекстах. В языке Perl, например, программа

```
$a = "3";
print $a . 7 . "\n"; # '.' это составление
print $a + 7 . "\n"; # '+' это сложение
```

будет печатать

```
37
10
```

Эта контекстная интерпретация похожа на неявное приведение типа данных, за исключением того, что нет понятия «естественного» типа, из которого должен быть преобразован объект; все различные возможные интерпретации считаются «одинаково естественными».

Легкий доступ к другим программам. Большинство языков программирования предоставляет возможность посылки из одной программы запроса на запуск другой программы. Этот запрос адресуется операционной системе. В скриптовых языках, однако, эти запросы более фундаментальны и имеют более развитую прямую поддержку. Язык Perl, например, предлагает более 100 встроенных команд для доступа к функциям ввода-вывода операционной системы, манипуляций с файлами и каталогами, управления процессами, доступа к базе данных и сокетам, взаимодействию и синхронизации процессов, защите и авторизации, службе времени и сетевому взаимодействию. Эти встроенные команды более просты для использования, чем соответствующие вызовы из библиотек на C-подобных языках.

Сложное сопоставление с образцом и обработка строк. Поскольку их предками являются обработчики текстов и генераторы отчетов, скриптовые языки предоставляют необычно развитые средства для сопоставления с образцом, поиска и обработки строк. Эти средства существенно упрощают текстовый ввод-вывод из внешних программ и основаны на расширенных регулярных выражениях.

Высокоуровневые типы данных. Высокоуровневые типы данных, такие как множества, мультимножества, словари, списки и кортежи, широко представлены в стандартных библиотечных пакетах традиционных языков программирования. Некоторые языки (особенно C++) позволяют пользователям переопределять стандартные инфиксные операции так, чтобы сделать обработку этих категорий столь же удобной, как и в случае примитивных данных аппаратного уровня. Скриптовые языки идут на один шаг дальше, встраивая высокоуровневые типы в синтаксис и семантику самого языка. Для большинства скриптовых языков, например, совершенно обычно наличие массивов, которые индексируются строками символов, с базовой реализацией на основе хеш-таблиц. Для памяти обеспечивается сборка мусора. Наиболее быстрые изменения в языках программирования сегодня происходят именно в скриптовых языках. Это объясняется несколькими причинами: непрерывным ростом веб-сети, динамизмом сообщества разработчиков ПО с открытым исходным кодом и сравнительно низкими затратами на создание нового языка скриптов. Там, где компилируемый язык промышленного качества (Java или C#) требует многолетних затрат и очень большой группы разработчиков, один талантливый разработчик, работая в одиночку, может создать пригодную для использования реализацию нового языка скриптов за год.

Отчасти благодаря этим быстрым изменениям последние скриптовые языки интегрировали в себя новейшие концепции в проектировании языков. Ruby, например, имеет универсальную объектную модель (подобно Smalltalk), настоящие итераторы (как в Clu), сечения массивов (как в Fortran 90), структурированную обработку исключений, множественное присваивание и рефлексию. Python также предоставляет некоторые из этих характеристик, вместе с анонимными функциями первого класса и представлениями списков в стиле языка Haskell.

Язык Perl

Язык Perl изначально был комбинацией SH и AWK. С тех пор Perl существенно расширился и в настоящее время является мощным (хотя и несколько примитивным) языком программирования. Конечно, его называют скриптовым языком, но на самом деле Perl больше похож на императивный язык, так как перед выполнением его программы всегда компилируются (в код на некотором промежуточном языке). Кроме того, его конструкции рассчитаны на применение в самых различных областях вычислительных задач. Perl имеет ряд интересных особенностей, лишь немногие из которых упоминаются в этой главе [43].

Для переменных в Perl характерны статическая типизация и неявное объявление. Существуют три категории, обозначаемые первым символом имени переменных. Все имена скалярных переменных начинаются со знака доллара ($), все имена массивов начинаются со знака (@), а все хеш-имена начинаются со знака процента (%). Это соглашение делает имена переменных в программах более читабельными, чем в любом другом языке программирования.

Perl включает в себя большое количество неявных переменных. Некоторые из них используются для хранения параметров Perl, таких как особая форма *символа новой строки* или символов, которые используются в процессе реализации. Неявные (скрытые) переменные обычно используются в качестве параметров по умолчанию для встроенных функций (и операндов по умолчанию для некоторых операций). Неявные переменные имеют характерные, хотя и загадочные имена, например `$!` и `@_`. Имена неявных пользовательских переменных могут относиться ко всех трем категориям, в то время как `$!` всегда считается «скалярным» именем.

Массивы в Perl имеют две особенности, которые отличают их от массивов в обычных императивных языках. Во-первых, для них характерна динамическая длина, а это означает, что они могут увеличиваться и уменьшаться по мере необходимости во время исполнения программы. Во-вторых, массивы могут быть *разреженными*, то есть между элементами могут быть промежутки. На хранение промежутков память не расходуется, а оператор цикла для массивов `foreach` пропускает отсутствующие элементы.

Perl включает ассоциативные массивы, называемые хешами, так как их элементы записываются в память и извлекаются из нее с помощью функций хеширования. Это структуры данных, которые индексируются строками и неявно управляются хеш-таблицами. Система Perl применяет хеш-функции и увеличивает размер структуры в случае необходимости. Ассоциативные массивы часто называют *массивами с адресацией по содержанию*, поскольку для доступа к информации можно пользоваться не только индексами массива, но и содержанием элемента.

Perl является мощным, но несколько опасным языком. Его скалярный тип поддерживает как строки, так и числа, которые обычно хранятся в формате с плавающей точкой двойной точности. В зависимости от контекста числа могут неявно преобразовываться в строки, и наоборот. Если строка применяется в числовом контексте и не может быть конвертирована в число, то используется ноль, а сообщение об ошибке для пользователя не формируется. Это соглашение может приводить к ошибкам, которые не обнаруживаются компилятором или системой периода выполнения. Индексация массивов также не проверяется, поскольку для массива не задается диапазон индексов. Ссылки на несуществующие элементы возвращают значение `undef`, которое интерпретируется как ноль в числовом контексте. Таким образом, ошибки в доступе к элементам массива здесь не обнаруживаются.

Первой программой, написанной на Perl, была утилита для обработки текстовых файлов в среде UNIX. Язык и сейчас используется для создания утилит системного администрирования UNIX. С появлением Всемирной паутины Perl стал применяться для задания сценариев CGI (интерфейса общего шлюза — Common Gateway Interface), хотя сейчас он достаточно редко используется для этих целей. Perl применяется и как язык общего назначения для различных приложений, таких как вычислительная биология и искусственный интеллект.

Ниже приведен пример программы на языке Perl:

```
# Программа на языке Perl
# Ввод:  Целая переменная $listlen – количество вводимых целых чисел,
#        значение $listlen должно быть меньше 100.
# Вывод: Количество введенных чисел со значениями больше среднего,
#        среднее значение рассчитывается для всех введенных чисел.
($sum, $result) = (0, 0);
```

продолжение ⇩

```
$listlen = <STDIN>;
if (($listlen > 0) && ($listlen < 100)) {
# Занесение чисел в массив и расчет суммы
    for ($counter = 0; $counter < $listlen; $counter++) {
        $intlist[$counter] = <STDIN>;
    } #- end of for (counter ...
# Вычисление среднего
    $average = $sum / $listlen;
# Расчет количества введенных чисел со значениями > среднего
    foreach $num (@intlist) {
        if ($num > $average) { $result++; }
    } #- end of foreach $num ...
# Печать результата
    print "Количество чисел со значениями > среднего : $result \n";
} #- end of if (($listlen ...
else {
    print "Ошибка – длина списка некорректна \n";
}
```

Язык JavaScript

Огромную популярность веб-приложения снискали в середине 90-х годов, после появления первых графических браузеров. Крайне необходимыми стали вычисления, связанные с документами HTML, которые сами по себе были целиком статичны. Были найдены два решения. Вычисления на стороне сервера обеспечивались интерфейсом общего шлюза CGI, который позволил документам HTML запрашивать исполнения программ на сервере, а результаты таких вычислений возвращались в браузер в виде HTML-документов. В свою очередь, апплеты Java сделали реальными вычисления на стороне браузера. Чуть позже были предложены новые технологии на основе скриптовых языков.

Язык JavaScript (Фланаган, 2002) первоначально разработан Бренданом Эйкхом (Brendan Eich) в корпорации Netscape. Вначале он назывался Mocha. Позднее язык был переименован в LiveScript. В конце 1995 года LiveScript стал совместной разработкой Netscape и Sun Microsystems, получившей окончательное название JavaScript. JavaScript существенно развился при переходе от версии 1.0 к версии 1.5. Было добавлено много новых функций и возможностей. Стандарт для языка JavaScript был разработан в конце 90-х Европейской ассоциацией производителей компьютеров (ECMA) как ECMA-262. Этот стандарт также был одобрен Международной организацией по стандартизации как ISO-16262. Версия от Microsoft называется JScript.NET.

Хотя интерпретатор JavaScript может быть встроен в различные приложения, наиболее часто он встраивается в веб-браузеры. Код JavaScript встраивается в HTML-документы и интерпретируется браузером при отображении документов. В веб-программировании JavaScript преимущественно используется для проверки данных в формах ввода и создания динамических HTML-документов. Кроме того, JavaScript в настоящее время используется в среде разработки веб-приложений Rails.

Несмотря на свое название, JavaScript похож на Java лишь на уровне синтаксиса отдельных конструкций. Java является сильно типизированным языком, а JavaScript поддерживает динамическую типизацию. Символьные строки и массивы в JavaScript имеют динамическую длину. В связи с этим достоверность индексов массива не про-

веряется, хотя в Java такие проверки осуществляются. Java полностью поддерживает объектно-ориентированное программирование, а JavaScript не поддерживает ни наследование, ни динамическое связывание при вызове методов [59].

Одной из наиболее важных областей применения JavaScript является динамическое создание и модификация HTML-документов. JavaScript определяет иерархию объектов, которая соответствует иерархической модели документа HTML, заданной объектной моделью документа Document Object Model. Элементы документа HTML доступны через эти объекты, обеспечивая базис для динамического управления содержанием документов. Программы, написанные на языке JavaScript, встраиваются непосредственно в HTML-документы с помощью парного тега `<script>`.

Представим сценарий на языке JavaScript. Обратите внимание: предполагается, что этот сценарий будет вызываться из HTML-документа и интерпретироваться веб-браузером.

```
// example.js
// Ввод:  Целая переменная $listlen – количество вводимых целых чисел,
//        значение $listlen должно быть меньше 100.
// Вывод: Количество введенных чисел со значениями больше среднего,
//        среднее значение рассчитывается для всех введенных чисел.
var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;
listLen = prompt (
    "Введите длину списка чисел", "");
if ((listLen > 0) && (listLen < 100)) {
// Занесение чисел в массив и расчет суммы
    for (counter = 0; counter < listLen; counter++) {
        intList[counter] = prompt (
          "Наберите следующее число", "");
        sum += parseInt(intList[counter]);
    }
// Вычисление среднего
    average = sum / listLen;
// Расчет количества введенных чисел со значениями > среднего
    for (counter = 0; counter < listLen; counter++)
        if (intList[counter] > average) result++;
// Отображение результата
    document.write("Количество чисел со значениями > среднего : ",
result, "<br />");
} else
    document.write(
        " Ошибка – длина списка некорректна <br />");
```

В последнем операторе `document.write` выводимая строка закачивается тегом HTML, обеспечивающим переход на новую строку `<br />`.

Язык PHP

Создателем языка PHP (Converse и Park, 2000) является Расмус Лердорф, член группы Apache, он разработал язык в 1994 году. Его первоначальная мотивация заключалась в предоставлении инструмента для отслеживания посетителей своего личного веб-сайта. В 1995 году Лердорф разработал пакет под названием Personal Home Page Tools, который стал первой публично распространяемой версией PHP. В оригинале PHP — аббревиатура от фразы Personal Home Page [59].

Несколько позже аббревиатуру PHP стали расшифровывать как «PHP Hypertext Preprocessor». Язык PHP сейчас разрабатывается, распространяется и поддерживается как продукт с открытым исходным кодом. Процессоры PHP располагаются на большинстве веб-серверов.

PHP является скриптовым языком серверной стороны, встраиваемым в HTML и специально предназначенным для веб-приложений. Код PHP интерпретируется на веб-сервере, когда HTML-документ, в котором он встроен, запрашивается браузером. В результате интерпретации программного кода PHP обычно генерируется код HTML, который заменяет код PHP в документе HTML. Таким образом, веб-браузер не видит кода PHP.

В PHP реализованы лучшие решения многих языков, таких как C и Perl. Он похож на JavaScript своим синтаксисом, динамическим характером своих строк и массивов, а также использованием динамической типизации. Массив в PHP заимствует черты массива JavaScript, а также хеша из языка Perl.

Оригинальная версия PHP игнорировала объектно-ориентированный стиль программирования, но его поддержка была добавлена во второй версии. Однако в PHP отсутствуют абстрактные классы и интерфейсы, деструкторы, нет контроля доступа к элементам класса.

PHP обеспечивает простой доступ к HTML-формам данных, так что обработка форм в PHP очень проста. PHP предоставляет разработчику мощные средства для работы с различными базами данных. Это делает его полезным языком для создания программ, которым нужен веб-доступ к базам данных.

Язык Python

Python (Лутц и Ашер, 2004) является относительно новым объектно-ориентированным и интерпретируемым языком скриптов. Его первоначальное проектирование выполнил Гвидо ван Россум в математическом центре Stichting (Нидерланды, начало 90-х). Развитие языка в настоящее время поддерживается фондом Python Software. Python используется для тех же типов задач, что Perl. В их число входят системное администрирование, программирование CGI и другие относительно небольшие вычислительные задачи. Python является системой с открытым исходным кодом и доступен для наиболее популярных платформ. Реализации Python размещены на сайте *www.python.org*, который предлагает также обширную информацию о Python [22].

Синтаксис языка Python отличается от синтаксиса обычного языка. Типы элементов данных проверяются, но типизация динамическая. Вместо массивов Python включает в себя три вида структур данных: списки, неизменяемые списки, которые называются кортежами, и хеши, именуемые словарями. Предлагается обширный набор методов для списков: добавление, вставка, удаление и сортировка, а также набор методов для словарей, таких как ключи, значения, копирование и `has_key`. Python также поддерживает списки в стиле языка Haskell.

Python считается объектно-ориентированным языком и включает в себя возможности сопоставления с шаблоном, присущие языку Perl, предусмотрена в нем и обработка исключений. Для удаления ненужных объектов применяется сборка мусора.

Поддержка CGI программирования и обработки форм, в частности, обеспечивается CGI-модулем. Имеются также модули с поддержкой куки, механизмы сетевого взаимодействия и доступа к базе данных.

Python включает в себя поддержку параллелизма на основе потоков, а также поддержку сетевого программирования с использованием сокетов. Дополнительно он предлагает более широкую поддержку функционального программирования по сравнению с другими нефункциональными языками.

Одной из наиболее интересных особенностей Python является то, что он может быть легко расширен любым пользователем. Модули, которые поддерживают расширения, могут быть написаны на любом компилируемом языке. Расширения могут добавлять функции, переменные и типы объектов. Эти расширения реализуются в виде дополнений к интерпретатору языка Python.

Язык Ruby

Язык Ruby был разработан Юкихиро Мацумото (он более известен как Мац) в начале 90-х годов и реализован в 1996 году. С тех пор он постоянно развивается. Мотивацией для Ruby была неудовлетворенность его разработчика языками Perl и Python. Несмотря на то что Perl и Python поддерживают объектно-ориентированное программирование, ни один из них не является чисто объектно-ориентированным языком в том смысле, что каждый из них имеет примитивные (не объектные) типы для каждой поддерживаемой функции [37].

Ruby — это полностью объектно-ориентированный язык в том понимании, что и язык Smalltalk. Каждый элемент данных рассматривается как объект, и все операции осуществляются через вызовы методов. Операции в Ruby считаются лишь синтаксическим механизмом, указывающим метод, требуемый для соответствующей операции. Поскольку все операции являются методами, они могут переопределяться. Все классы, предопределенные и определяемые пользователем, могут иметь подклассы.

Классы и объекты в Ruby являются динамическими в том отношении, что методы к ним могут добавляться динамически. Это означает, что и классы, и объекты могут иметь различные наборы методов в разные моменты периода выполнения. Таким образом, различные экземпляры одного и того же класса могут вести себя по-разному. В определение класса могут быть включены коллекции методов, данных и констант.

Синтаксис Ruby похож на синтаксис языков Eiffel и Ada. Здесь нет необходимости объявлять переменные, так как используется динамическая типизация. Область видимости переменной указывается ее именем: переменная, имя которой начинается с буквы, имеет локальную область, переменная, имя которой начинается с `@`, является экземплярной переменной, если же имя начинается с `$`, переменная считается глобальной. В Ruby присутствует ряд особенностей языка Perl, в том числе скрытые (неявные) переменные с глупыми именами, такими как `$_`.

Как и в случае с языком Python, любой пользователь может расширить и/или изменить Ruby. Ruby интересен в культурном отношении, поскольку это первый язык программирования, созданный в Японии, который достаточно широко распространен и используется во всем мире.

Язык Lua

Язык Lua был разработан в начале 90-х Роберто Иерусалимски, Вальдемаром Келес и Луисом де Фигейреду в Католическом университете Рио-де-Жанейро (Бразилия). Lua в переводе с португальского означает «луна». Lua обычно называют мультипарадигменным языком. В частности, он обеспечивает процедурное и функциональное программирование. Lua содержит небольшой набор базовых механизмов, которые могут быть расширены для решения различных задач, а не набор сложных жестких спецификаций, обеспечивающих программирование в единой парадигме [69]. Среди языков, которые повлияли на его разработку, следует назвать Scheme, Icon и Python.

По возможностям, идеологии и реализации язык ближе всего к JavaScript, однако Lua отличается более мощными и гораздо более гибкими конструкциями. Ключевой особенностью Lua является расширяемая семантика. Хотя Lua не содержит понятия класса и объекта в явном виде, механизмы объектно-ориентированного программирования с поддержкой прототипов (включая множественное наследование) легко реализуются с использованием мета-таблиц, которые также делают возможной перегрузку операций и т. д.

Примеры программного кода на Lua

Классическая программа `hello world` имеет вид:

```
print("Hello World!")
```

Синтаксис комментариев близок к языку Ada:

```
-- Комментарии в Lua начинаются с двойного дефиса и продолжаются до конца строки.
--[[ Многострочные текстовые литералы и комментарии
     записывают в двойных квадратных скобках. ]]
```

Вычисление факториала:

```
function factorial(n)
  if n == 0 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
```

Цикл с предусловием:

```
local i = 1
while a[i] do
    print(a[i])
    i = i + 1
end
```

Цикл с постусловием:

```
-- печатать первую непустую строку ввода
repeat
    line = os.read()
until line ~= ""
print(line)
```

Цикл со счетчиком:

```
for i = 1, 7 do
     -- Операторы
end
```

Функции языка Lua

Как и в языке Scheme, функции в Lua являются величинами первого класса, то есть ими можно манипулировать точно так же, как переменными: передавать и получать как аргументы и т. д.

Работа с функциями как с объектами первого класса демонстрируется в следующем примере, в котором изменяется поведение функции `print`:

```
do
  local oldprint = print   -- Сохраняем текущую функцию print как oldprint
  function print(s)        -- Переопределяем функцию print
    if s == "foo" then
      oldprint("bar")
    else
      oldprint(s)
    end
  end
end
```

Любой будущий вызов `print` теперь будет перенаправлен к будущей функции, и благодаря поддержке в Lua лексического контекста старая функция `print` будет доступна только посредством новой, измененной функции `print`.

Кроме того, Lua поддерживает замыкания. Замыкание — это особый вид функции. Она определена в теле другой функции и создается каждый раз во время ее выполнения. В записи это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

В случае замыкания ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу и переменные вышли из области видимости.

Замыкание связывает код функции с ее лексическим окружением (местом, в котором она определена в коде). Лексические переменные замыкания отличаются от глобальных переменных тем, что они не занимают глобальное пространство имен. От переменных в объектах они отличаются тем, что привязаны к функциям, а не объектам.

Приведем пример замыкания:

```
function makeaddfunc(x)
   -- Возвращает новую анонимную функцию, которая добавляет x к аргументу
   return function(y)
    -- Когда мы ссылаемся на переменную x, которая вне текущей области,
    -- и время жизни которой меньше, чем у этой анонимной функции,
    -- Lua создает замыкание.
    return x + y
  end
end
plustwo = makeaddfunc(2)
print(plustwo(5)) -- Выводит 7
```

Каждый раз, когда вызывается `makeaddfunc`, создается новое замыкание для переменной `x`, так что каждая возвращаемая анонимная функция будет ссылаться на свой параметр `x`. Как и у любого другого объекта Lua, временем жизни замыкания управляет сборщик мусора.

Перечисленные возможности позволяют применять язык Lua для функционального программирования.

Подобно таким скриптовым языкам, как Python и Ruby, Lua допускает параллельное присваивание. В простейшем случае это позволяет писать выражения вида:

```
x, y = y, x
```

Типы данных

Lua представляет собой язык с динамическим определением типов данных. Переменная языка может содержать значения любого типа. Все значения в Lua могут храниться в переменных, использоваться в качестве аргументов при вызове функций и возвращаться в виде результата их выполнения.

В Lua имеется восемь основных типов данных:

1. `nil` (неопределенный);
2. `boolean` (логический);
3. `number` (числовой);
4. `string` (строковый);
5. `function` (функция);
6. `userdata` (пользовательские данные);
7. `thread` (поток);
8. `table` (таблица).

`Nil` — это тип с пустым значением `nil`, главное свойство которого — отличаться от всех остальных значений и обозначать отсутствие пригодного значения. К типу `boolean` относятся значения `false` и `true`. Значения `nil` и `false` считаются ложными, любое другое значение считается истинным. Тип `number` определяет обычно вещественные числа (`double`), но внутреннее представление чисел можно изменить при сборке интерпретатора. Тип `string` обозначает массивы символов. Строки Lua могут содержать любые 8-битовые символы, включая ноль (`'\0'`). Строки неизменяемы.

Подобно языку Scheme, Lua имеет только одну структуру данных, хотя в случае Lua это таблица. Таблицы Lua являются расширением ассоциативного массива из PHP, который относится к категории массивов традиционных императивных языков. Таблицы считаются самым важным типом данных в Lua и рассматриваются как фундамент для строительства пользовательских типов данных, таких как массивы, списки, множества. Таблица в Lua представляет собой набор пар — (ключ, значение). Ключом может выступать значение любого типа, кроме `nil`. Например, с помощью таблицы можно создать матрицу:

```
mt = {} – создание матрицы
for i=1,N do
    for j=1,M do
        mt[(i-1)*M + j] = 0
    end
end
```

Ссылки на элементы таблицы могут осуществляться в форме ссылок на традиционные массивы, ассоциативные массивы или записи. Поскольку функции относятся к первому классу, они могут храниться в таблицах, и такие таблицы могут служить пространствами имен.

Остальные характеристики

Lua использует сборку мусора для объектов, которые размещены в куче. Повторим, что в нем реализована динамическая типизация, как и в большинстве других скриптовых языков.

Lua является относительно небольшим и простым языком, задействует только двадцать одно зарезервированное слово. Философия проектирования языка была ориентирована на обеспечение самых необходимых и простых способов расширения языка для различных областей применения. Большая часть его расширений строится на основе табличной структуры данных, которая может быть специализирована с помощью концепции мета-таблицы Lua.

Механизм мета-таблиц дает большие возможности по настройке уникального поведения для таблиц Lua. В следующем примере демонстрируется «бесконечная» таблица. Для любого n fibs[n] даст n-е число Фибоначчи с использованием мемоизации.

```
fibs = { 1, 1 }                  -- Таблица, начальные значения fibs[1] и fibs[2].
setmetatable(fibs, {
  _index = function(name, n)     -- Вызов функции, если fibs[n] не существует.
    name[n] = name[n - 1] + name[n - 2]  -- Расчет и мемоизация fibs[n].
    return name[n]
  end
})
```

ПРИМЕЧАНИЕ

Мемоизация — специальная оптимизационная методика, которая позволяет увеличить скорость выполнения компьютерных программ. Данная методика заключается в том, чтобы исключить повторное вычисление результатов предыдущих вызовов.

Lua удобно использовать как расширение скриптового языка для других языков. Как и в ранних реализациях Java, текст Lua транслируется в промежуточный код, а затем интерпретируется. Он легко может быть встроен в другие системы, в частности, из-за небольшого размера его интерпретатора, который составляет всего около 150К байт.

В течение 2006–2007 годов популярность Lua росла очень быстро, в частности, благодаря использованию в игровой индустрии. В цепочке скриптовых языков, которые появились за последние 20 лет, уже создано несколько широко используемых языков. Lua, последний среди них, быстро становится одним из весьма популярных языков.

Гибридные языки разметки/программирования

Гибридный язык разметки/программирования является языком разметки, в котором некоторые элементы могут задавать действия по программированию, такие как управление потоком действий и вычисления.

Язык XSLT

Расширяемый язык разметки (XML) является языком мета-разметки. Такой язык используется для определения языков разметки. Производные от XML языки разметки используются для определения документов данных, называемых XML-документами. Хотя XML-документы читабельны для человека, они обрабатываются с помощью компьютеров. Иногда эта обработка состоит только из преобразований в формы, которые могут эффективно отображаться или распечатываться. Во многих случаях это преобразования в формы HTML, которые могут отображаться веб-браузером. В других случаях данные в документе обрабатываются так же, как и другие формы файлов с данными.

Преобразования XML-документов в XML- и HTML-документы определяются другим языком разметки, расширяемым языком стилей для преобразований XSLT. XSLT может указывать операции подобные программированию. Таким образом, XSLT является гибридным языком разметки/программирования [7, 35].

Спецификация XSLT гласит, что это язык для преобразования одних XML-документов в другие XML-документы. Именно такой и была изначальная идея XSLT. Тем не менее по мере разработки язык перерос ее и теперь, как утверждает редактор новой версии языка Майкл Кей (Michael Kay), XSLT — это язык для преобразования структуры документов.

В преобразовании участвуют три документа:

- входной документ, который подвергается преобразованию (XML-документ);
- преобразователь (документ XSLT, который описывает само преобразование);
- выходной документ, который является результатом преобразования.

Преобразователь задает правила трансформации входного документа в выходной.

Выполнением преобразований занимаются специальные программы — процессоры XSLT. Процессор получает входной документ и преобразователь и, применяя правила преобразования, генерирует выходной документ.

По сути процессор оперирует не самими документами, а древовидными моделями их структур — именно структурными преобразованиями занимается XSLT, оставляя за кадром синтаксис, который эти структуры выражает.

Несмотря на то что для XSLT совершенно неважно, в каком виде находятся документы изначально (главное — чтобы была структура, которую можно преобразовать), абсолютное большинство процессоров XSLT может работать с документами, которые физически записаны в файлах. В этом случае процесс обработки делится на три этапа.

- Этап разбора (парсинга) документа. Здесь XSLT-процессор разбирает входящий документ и документ-преобразователь, создавая для них древовидные структуры данных.
- Этап преобразования. К дереву входного документа применяются правила, описанные в преобразователе. В итоге процессор создает дерево выходного документа.
- Этап сериализации. Для созданного дерева генерируется сущность в определенной физической форме.

Процессоры выполняют каждый из трех этапов (получают входные документы и выдают результат их трансформации), но областью применения XSLT является лишь второй этап — этап преобразования.

Сериализация ориентирована на создание физической интерпретации конечного дерева. Текущая версия языка поддерживает три способа сериализации: XML, HTML и текст. Каждый из этих способов учитывает синтаксис целевого физического формата и позволяет получить документ требуемого вида.

Преобразователь может указывать способ сериализации, однако непосредственный контроль над синтаксисом физического документа сильно ограничен. Преобразования совершенно сознательно отделены от синтаксической обработки, ведь их задачей являются структурные трансформации, а не работа с физическим синтаксисом. Благодаря такому разделению многие процессоры производят разбор и сериализацию с помощью внешних приложений, что способствует универсальности XSLT. Ведь для каждого этапа преобразования можно применить наиболее подходящий инструмент.

С точки зрения языка XSLT задачу преобразования можно разделить на три подзадачи:

- обращение к преобразуемому объекту;
- создание результата преобразования;
- связывание первых двух действий для осуществления преобразования.

Первая подзадача обеспечивает получение информации о входном документе, его структуре. Здесь применяется язык XPath — язык путей в XML-документах.

Вторая и третья подзадачи решаются программой на XSLT, которую мы назвали преобразователем. Синтаксически преобразователь — это XML-документ.

В отличие от традиционных языков программирования, преобразование в XSLT не описывается последовательностью действий, выполняемых для достижения результата. Преобразование указывается в виде набора шаблонных правил, каждое из которых определяет процедуру обработки определенной части документа. Иначе говоря, преобразователь в XSLT объявляет, декларирует правила преобразования — правила, применяя которые к входному документу, XSLT-процессор генерирует выходной документ.

Рассмотрим пример XSLT-преобразования. Поставим задачу преобразования входного документа

```
<msg>Hello, world!</msg>
```

в выходной документ вида:

```
<message>Hello, world!</message>
```

Суть преобразования: если в документе встретится элемент `msg`, создать в выходном документе элемент `message` и включить в него содержимое элемента `msg`.

Эта задача решается следующей программой на XSLT:

```
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="msg">
        <message>
            <xsl:value-of select="."/>
```

продолжение ⇩

```
        </message>
    </xsl:template>
</xsl:stylesheet>
```

Корневым элементом программы является элемент `xsl:stylesheet`, обозначающий преобразование. Атрибут `version` указывает на использованную версию языка XSLT; здесь же объявляется пространство имен с префиксом `xsl`, которому соответствует URL `"http://www.W3.org/1999/XSL/Transform"`. Все элементы преобразования из этого пространства имен будут восприняты процессором как принадлежащие языку XSLT.

Элемент `xsl:stylesheet` содержит единственный дочерний элемент `xsl:template`, задающий правило преобразования. Атрибут `match` указывает, что данное правило должно обрабатывать элемент `msg`. Содержимое `xsl:template` является телом шаблона. Тело выполняется тогда, когда сам шаблон применяется к некоторой части документа. В данном случае тело шаблона будет выполнено, когда само правило будет применяться к элементу `msg`.

В теле шаблона записан элемент `message`. Это простой литеральный элемент результата. Он не принадлежит пространству имен XSLT и поэтому при обработке будет просто скопирован в результирующий документ. Содержимое этого элемента будет обработано и включено в сгенерированную копию.

Содержимым элемента `message` является элемент `xsl:value-of`, который, в отличие от `message`, принадлежит XSLT. Элемент `xsl:value-of` вычисляет XPath-выражение, заданное в его атрибуте `select`, и возвращает результат этого вычисления. XPath-выражение, ".", указанное в `select`, возвращает ту часть узла дерева, которая обрабатывается в данный момент, иначе говоря элемент `msg`.

В отличие от синтаксиса языка XPath (представлен лаконичным выражением "."), синтаксис языка XSLT полностью воспроизводит синтаксис XML.

В описанном преобразовании участвовала и третья синтаксическая конструкция — паттерн XSLT. Паттерн `msg`, записанный в атрибуте `match` элемента `xsl:template`, задает ту часть XML-документа, которая должна быть обработана этим правилом. Синтаксически паттерны считаются XPath-выражениями (но не наоборот), однако смысл у них другой. XPath-выражения вычисляются и возвращают результат, паттерны же просто устанавливают соответствие некоторому образцу. В нашем преобразовании паттерн `msg` указывает, что шаблон должен обрабатывать только элементы `msg` и никакие другие.

Каждое из шаблонных правил может вызывать другие шаблонные правила — в этом случае результат выполнения вызванных шаблонов включается в результат выполнения шаблона, который их вызывал. Для демонстрации этого приема модифицируем программу так, чтобы результат возвращался в виде HTML-документа:

```
<xsl:stylesheet
    version="l. 0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <head>
            <title>Message</title>
            </head>
```

```
            <body>
                <xsl:apply-templates select="msg"/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="msg">
            <b>
                <xsl:value-of select="."/>
            </b>
    </xsl:template>
</xsl:stylesheet>
```

В этот преобразователь добавлено еще одно шаблонное правило:

```
<xsl:template match="/">
    <html>
        <head>
            <title>Message</title>
        </head>
        <body>
            <xsl:apply-templates select="msg"/>
        </body>
    </html>
</xsl:template>
```

Данное правило задает обработку корневого узла — в атрибуте `match` указан патерн `"/"`, что соответствует корню документа. Шаблон создает элементы `html`, `head`, `title`, `body` и в последний включает результат применения шаблонов к элементу `msg`.

Выходной документ, полученный в результате применения этого преобразователя, примет вид:

```
<html>
    <head>
        <title>Message</title>
    </head>
    <body>
        <b>Hello, world!</b>
    </body>
</html>
```

Отметим, что процессор скопировал все элементы, не принадлежащие XSLT, не изменяя их, а элемент `xsl:apply-templates` выполнил подстановку, применив шаблон к элементу `msg` и включив в `body` результат (выделен полужирным шрифтом).

Возможность вызова одних правил из других, а также наличие в XSLT таких управляющих конструкций, как `xsl:if`, `xsl:choose` и `xsl:for-each`, обеспечивает реализацию очень сложной логики преобразования. XSLT позволяет определить простые правила обработки отдельных частей преобразования, связав эти правила логикой взаимных вызовов и управляющих конструкций.

Таким образом, с каждым шаблоном в документе XSLT ассоциированы инструкции преобразования, которые определяют, как сопоставленные данные должны быть преобразованы перед их размещением в выходном документе. Можно сказать, что шаблоны (и ассоциированная с ними обработка) выступают в качестве подпрограмм, которые «выполняются», когда процессор XSLT находит сопоставление с шаблоном в данных документа XML.

Язык JSP

Основной частью стандартной библиотеки тегов серверных страниц Java (JSTL) является еще один гибридный язык разметки/программирования, хотя его форма и цели отличны от XSLT. Прежде чем обсуждать JSTL, необходимо ввести понятия сервлетов и Java Server Pages (JSP). Сервлет является экземпляром класса Java, который находится и выполняется на веб-серверной стороне. Выполнение сервлета, запрашиваемое документом разметки, отображается веб-браузером. Выводы сервлета возвращаются запрашивающему браузеру в форме документа HTML. Программа, которая выполняется веб-сервером, называется контейнером сервлетов и контролирует выполнение сервлетов. Сервлеты обычно используются для обработки форм и доступа к базе данных.

JSP представляет собой набор технологий, предназначенных для поддержки динамических веб-документов, а также обеспечивающих иные потребности в обработке веб-документов [59]. Если документ JSP, который часто является комбинацией кодов HTML и Java, запрашивается браузером, процессорная программа JSP, которая находится в веб-серверной системе, преобразует документ в сервлет. Встроенный в документ Java-код копируется в сервлет. Простой код HTML копируется в операторы печати Java, что рассматривается как вывод. Сервлет производится процессором JSP, который запускается контейнером сервлетов. JSTL определяет набор элементов действия для XML, которые управляют обработкой документа JSP на веб-сервере. Эти элементы имеют тот же вид, что и другие элементы HTML и XML. Наиболее часто используется такое управляющее действие JSTL, как if, определяющее в качестве атрибута логическое выражение. Содержанием элемента if (текстом между открывающим тегом `<if>` и закрывающим тегом `</if>`) является код HTML, который будет включен в выходной документ только в том случае, если логическое выражение будет истинным. Элемент if подобен препроцессорной команде #if для языков C/C++. Контейнер JSP обрабатывает JSTL-части JSP-документа подобно тому, как препроцессор C/C++ обрабатывает программы C и C++. Команды препроцессора представляют собой инструкции для препроцессора, указывающие, как из входного файла строится выходной файл. Кроме того, элементы управления действиями JSTL являются инструкциями для процессора JSP, определяющими порядок создания XML-файла выходных данных на основе входного XML-файла.

Чаще всего элемент if используется для проверки формы данных, представляемых пользователем браузера. Форма данных доступна процессору JSP и может быть проверена элементом if на предмет корректности данных. Если проверка не проходит, элемент if может вставить в выходной документ сообщение для пользователя об ошибке.

Для управления множественным выбором в JSTL могут использоваться choose, when и другие элементы. JSTL также включает в себя элемент forEach, который выполняет итерации по коллекциям, формируемым клиентами. Для управления итерациями элемент forEach может содержать атрибуты begin, end, а также step.

Контрольные вопросы и упражнения

1. Возьмите какую-нибудь программу, написанную вами на императивном языке (например, C или Pascal), и перепишите ее так, чтобы она стала функциональной.
2. Возможности языка C позволяют описать некоторое действие несколькими разными способами. Сколько различных операторов вы можете написать на C, чтобы прибавить единицу к переменной `x` (эквивалентных оператору `x = x + 1`)? Выясните преимущества и недостатки этого аспекта языка C.
3. Какова была основная область применения компьютеров во время создания языка Fortran?
4. Что общего у функциональных и логических языков? Чем они отличаются друг от друга?
5. Из чего состоит среда языка Smalltalk?
6. Какие основные идеи положены в основу объектно-ориентированного программирования?
7. Сравните императивные и скриптовые языки. Что в них общего? Чем они отличаются друг от друга?
8. Чего лишен язык HTML по сравнению с любым императивным языком? Какие его черты отсутствуют в любом императивном языке?
9. По какой причине языки XSLT и JSP называют гибридными? Обоснуйте их «гибридность», перечислив конкретные характерные свойства. Сравните эти языки друг с другом.

Глава 4

Выражения и присваивания в языках программирования

В этой главе рассматриваются разные формы записи выражений в языках программирования: префиксная, постфиксная, инфиксная и смешанная нотации. Обсуждается содержание и разновидности самой популярной и странной операции — присваивания. В завершение поясняется порядок вычисления операндов в выражении, всесторонне анализируется проблема побочных эффектов вычисления.

Нотации выражений

Любой язык программирования включает в себя средства описания данных и средства описания действий.

Средства описания данных декларируют элементы и типы данных, используемые в программе.

Средства описания действий указывают действия, исполняемые программой.

Простейшим средством описания действий является выражение. *Выражение* задает действия, которые вычисляют единственное значение. Состоит из констант, переменных, а также знаков операций и скобок. Элементы данных, к которым применяется операция, называют *операндами*.

Методы для описания синтаксиса выражений автоматически распространяются на остальные средства ЯП.

ЯП вносят некоторую специфику в запись математических выражений. Например, математическое выражение

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

на языке FORTRAN записывается в виде

```
(-b + sqrt(b * b - 4.0 * a * c))/(2.0 * a)
```

Обычно в ЯП используется смесь *нотаций* (*способов записи выражений*): большинство операций записывается между операндами, однако некоторые операции записываются перед их операндами, другие — после их операндов. В примере

операции +, –, * и / записаны между их операндами, однако минус записан перед *b* (в сегменте $-b$), а *sqrt* записана перед выражением, к которому она применяется.

Различают три нотации: инфиксную, префиксную и постфиксную [3, 31, 32, 84].

В качестве примеров будем рассматривать *бинарные* операции. Бинарной называют операцию, применяемую к двум операндам.

В инфиксной нотации бинарная операция записывается между ее операндами: $a + b$. В префиксной нотации операция записывается первой: $+ a\ b$, а в постфиксной нотации операция записывается последней: $a\ b\ +$.

Выражение может заключаться в скобки, это не влияет на его значение. Выражение *E*, как правило, имеет то же значение, что и (*E*).

Префиксную и постфиксную нотации называют бесскобочными. Причина — для определения операндов скобки не нужны.

Префиксная нотация выражения

Выражение в префиксной нотации определяется так:

- Префиксная нотация константы или переменной — это сама константа или переменная.
- Применение операции `op` к подвыражениям E_1 и E_2 записывается в виде `op` E_1 E_2.

Преимущество: префиксное выражение декодируется простым просмотром слева направо.

В такой нотации нет никакой неоднозначности и нет необходимости применять скобки для точного определения последовательности операций при вычислении выражения. Поскольку свободный от скобок способ записи изобрел польский математик Лукашевич, то для обозначения этой нотации применяется название *польская запись*.

Примеры

Сумме `x` и `y` соответствует префиксная запись `+ x y`.

Произведение `+ x y` и `z` задается префиксной записью `* + x y z`.

Значение префиксного выражения `+ 20 30` равно 50.

Выражение `* + 20 30 60` «сворачивается» до вида `* 50 60`, значение которого равно 3000.

Выражение `* 20 + 30 60` равно `* 20 90`, которое, в свою очередь, равно 1800.

Префиксная нотация может применяться только к операциям с фиксированным количеством `k` (`k`>0) операндов.

Количество операндов операции определяет ее *арность* (*местность*).

Применение операции op^k с арностью `k` ($k \geq 0$) к выражениям E_1, E_2, ..., E_k записывается как

$op^k\ E_1\ E_2\ \ldots\ E_k$

Здесь *i*-е выражение справа от op^k является ее *i*-м операндом (для $1 < i <$ k).

Специальные варианты префиксной нотации

```
read (a,b,c)      (max  x  y)
```

позволяют записывать операции с переменным количеством операндов.

Последний вариант называют *кембриджской польской записью*. В кембриджской польской записи операция и ее аргументы заключаются в скобки. Эта запись используется в языке LISP. В польской кембриджской записи обычное выражение `(a + b) × (c – a)` будет выглядеть следующим образом: `(* (+ a b)(– c a))`.

Для «ручного» преобразования в префиксную нотацию в исходном выражении ищется последняя (по выполнению) операция. Эта операция группирует (в префиксной нотации) свои операнды, затем выбирается предпоследняя операция и группирует свои операнды и т. д.

Примеры. Представим шаги преобразования выражения в префиксную форму. Шаги будем отделять друг от друга стрелкой следования →. Для ограничения области действия операций будем использовать вспомогательные скобки группировки:

```
a + b * c → + a (b * c) → + a * b c
a * b + c → + (a * b) c → + * a b c
```

С помощью префиксной нотации можно вычислить любое выражение за один просмотр. Однако надо знать заранее количество операндов у каждой операции. Как следствие, бинарное вычитание и унарный минус приходится обозначать различными символами. Впрочем, можно применить кембриджскую польскую запись со скобками.

Кроме экономии на отсутствии скобок в выражениях, префиксная нотация дает следующие дополнительные преимущества:

- Префиксная нотация удобна для представления операций с любым количеством операндов, в этом смысле она считается наиболее общей формой. Для записи любого выражения достаточно знать лишь одно синтаксическое правило. Например, чтобы создать любое выражение на функциональном языке, нужно освоить только кембриджскую польскую запись, и после этого вам будут понятны правила синтаксиса языка.
- Префиксное выражение достаточно легко вычислить механически, поэтому перевод префиксных выражений в простую последовательность кодов очень прост.

Простоту перевода в последовательность кодов проиллюстрируем следующим алгоритмом. Префиксное выражение `PE` можно вычислить с помощью стека:

1. Если очередной элемент в выражении `PE` обозначает символ операции, то заносим его в вершину стека. Заносим в счетчик операндов количество операндов, необходимых для данной операции.
2. Если следующий элемент является операндом, помещаем его в вершину стека.
3. Если верхние n элементов в стеке содержат набор операндов, достаточный для вычисления верхней n-арной операции, то эта операция применяется к операндам. Например, если символ умножения был последней операцией, записанной в стек, и в него же добавили два операнда. После выполнения операции ее символ и операнды из стека удаляются. Вместо них заносится результат.

Правда, здесь есть некоторая сложность — после занесения очередного операнда в стек приходится проверять, имеется ли достаточное количество операндов для применения соответствующей операции. Для отказа от такой проверки следует воспользоваться постфиксной нотацией.

Постфиксная нотация выражения

Выражение в постфиксной нотации определяется так:

- Постфиксная нотация константы или переменной — это сама константа или переменная.
- Применение операции op к подвыражениям E_1 и E_2 записывается в виде E_1 E_2 op.

  Постфиксную нотацию иначе называют *обратной польской записью*.

  Примеры

  Сумма x и y соответствует постфиксной записи x y +.

  Произведение x y + и z записывается в следующей форме: x y + z *.

  Значение выражения 20 30 + равно 50.

  Значение выражения 20 30 + 60 * равно 50 60 * или равно 3000.

  Значение выражения 20 30 60 + * равно 20 90 * или равно 1800.

Преимущество: постфиксные выражения могут вычисляться механически, с помощью структуры данных типа стек. Так как в постфиксной нотации операция записывается сразу за своими операндами, то при считывании символа операции ее операнды уже известны. Следовательно, вычисление постфиксного выражения EP выполняется по алгоритму:

1. Если следующий элемент является операндом, помещаем его в стек.
2. Если очередной элемент обозначает символ *n*-арной операции, то *n* ее операндов *должны находиться* в верхних ячейках стека. Заменяем эти операнды на результат выполнения соответствующей операции.

Словом, алгоритм вычислений прост и легко реализуем. В большинстве компиляторов этот алгоритм является основой для генерации кода, вычисляющего выражения. В ходе компиляции выражения часто переводятся в постфиксную нотацию, а для вычисления выражения генератор кода применяет описанный выше алгоритм.

Для «ручного» преобразования исходного выражения в постфиксную нотацию в нем ищется первая (по выполнению) операция. Эта операция группирует (в постфиксной нотации) свои операнды, затем выбирается вторая операция и группирует свои операнды и т. д.

Примеры

```
a + b * c → (a + b) c * → a b + c *
a * b + c → (a b *) + c → a b * c +
```

Формально подобное преобразование можно выполнять по *алгоритму Дейкстры*:

1. Проанализировать очередной элемент на входе.
2. Если это операнд, передать его на выход.
3. Если это «(», поместить ее в стек.
4. Если это операция, то:
   - если в вершине стека «(», поместить операцию в стек;
   - если у входной операции приоритет выше, чем у операции в вершине стека, поместить в стек новую операцию;
   - в противном случае — извлечь операцию из стека и отправить ее на выход, а затем повторить шаг 4.

5. Если это «**)**», извлекать из стека операции на выход до тех пор, пока в вершине стека не окажется «**(**». Извлечь «**(**» из стека и уничтожить.
6. Если на входе еще есть элементы, перейти к шагу 1.
7. Если на входе нет элементов, выталкивать из стека все операции на выход.

Инфиксная нотация выражения

В инфиксной нотации операции проставляются между их операндами. В сумме *a* и *b* операция сложения записывается между *a* и *b*.

Преимущество: легко читается человеком. Правда, хотя инфиксная нотация и широко распространена, ее использование создает определенные проблемы:

1. Поскольку инфиксная нотация определена лишь для бинарных операций, язык программирования не может использовать только одну эту форму записи для выражений, но обязательно должен сочетать ее с префиксной (постфиксной). Такое смешение делает, соответственно, компиляцию более сложной. Унарные операции и вызовы функций с несколькими аргументами не могут записываться в инфиксной нотации.
2. При появлении в выражении нескольких инфиксных операций для устранения неоднозначности приходится использовать скобки.

Действительно, как декодировать выражение $a + b * c$? Это сумма a и $b * c$? Или это произведение $a + b$ и c?

Для ответа на эти и другие вопросы вводятся понятия приоритетности и ассоциативности операций.

Приоритетность определяет порядок применения операции в выражении. Операция высокого приоритета выбирает свои операнды перед операцией низкого приоритета.

Рассмотрим выражение:

$$a + b * c.$$

В нашем выражении операция умножения выбирает свои операнды (и выполняется) перед операцией сложения.

Обычно операции с одинаковым приоритетом группируются слева направо:

$$4 - 2 + 1 = (4 - 2) + 1.$$

Точнее, так группируются *левоассоциативные операции* (+, –, *, /).

Ассоциативность задает порядок группировки в выражении операций с одинаковым приоритетом.

Левоассоциативная операция — подвыражения, содержащие ее множественные вхождения, группируются слева направо:

$$b * b - 4 * a * c = (b * b) - ((4 * a) * c).$$

Правоассоциативная операция — подвыражения, содержащие ее множественные вхождения, группируются справа налево:

$$2^{3^4} = 2^{\left(3^4\right)} = 2^{81}.$$

Как видим, правоассоциативной является операция возведения в степень.

Смешанная нотация

Операции, определяемые комбинацией символов, нельзя классифицировать в терминах префиксной, инфиксной и постфиксной нотации. Например, в выражении

```
if a > b then a else b
```

совместно используются ключевые (служебные) слова `if`, `then`, `else`.

Значащими компонентами этого выражения являются условие $a > b$ и выражения a и b.

Если значением условия является true, то значением выражения является a, в противном случае — b.

Говорят, что операция имеет *смешанную нотацию*, если ее значащие компоненты перемежаются со служебными словами (или символами).

Сравнение нотаций для записи выражений

Каждая из рассмотренных нотаций для выражений имеет свои достоинства и недостатки.

Инфиксная нотация обеспечивает довольно естественное представление для большинства арифметических, логических выражений и выражений сравнения. Правда, приходится использовать сложные неявные правила приоритетности и ассоциативности операций (они меняются от языка к языку), а также скобки. Кроме того, не двухместные операции все равно записываются в префиксной (постфиксной) нотации. Все эти проблемы существенно усложняет компиляцию инфиксных выражений.

Конечно, в инфиксной нотации необходимость учета приоритетности и ассоциативности операций можно свести к минимуму за счет скобок, но тогда понадобится огромное количество скобок. Проблема скобок также присутствует в кембриджской польской и в обычной математической префиксной записи.

В польской нотации вообще не применяются скобки, но нужно заранее знать количество операндов у каждой операции, что бывает трудно выполнить. Например, если речь идет об операциях, которые создаются программистами.

Префиксная и постфиксная нотации легко справляются с записью операций, имеющих различные количества операндов.

Наконец, человеку трудно читать сложные выражения, записанные в польской нотации.

Присваивание

Как правило, вычисленное значение выражения нужно где-то запоминать. Такую задачу решает операция присваивания. Присваивание — очень важное и популярное

действие в императивных языках программирования. Вместе с тем присваивание является не совсем обычной операцией. Рассмотрим ее специфику, предварительно договорившись о смене терминологии.

Будем теперь называть операнды выражения *объектами данных*. Это позволит учесть особенности их обработки в памяти компьютера. Упрощая, можно сказать: объект данных — это операнд, находящийся в ячейке памяти. Точнее, в ячейке находится его значение.

Присваивание — это элементарная операция, изменяющая связывание объекта данных со значением. Как ни странно, такое изменение считается побочным эффектом присваивания.

Поскольку присваивание является двухместной операцией, применительно к ее инфиксной нотации будем различать ее левый и правый операнды (аргументы).

В языке Pascal присваивание имеет такой смысл: скопировать в один (левый) объект данных значение, содержащееся в другом (правом) объекте данных такого же типа, но не возвращать явный результат. При этом изменение значения левого объекта считается неявным результатом (побочным эффектом) присваивания.

Несколько забегая вперед, условимся описывать двухместные операции в следующем формате:

```
символ операции: тип первого аргумента × тип второго аргумента → тип результата
```

ПРИМЕЧАНИЕ

Типом называется существенная характеристика объекта данных. Она задает допустимые значения, операции и способ размещения объекта в памяти. Подробно тип данных обсуждается в главе 8.

Соответственно, спецификация операции присваивания в языке Pascal примет вид:

```
присваивание (:=) : тип × тип → пустой тип (void)
```

В языке C смысл у присваивания несколько иной: скопировать в левый объект данных значение, содержащееся в правом объекте данных такого же типа, а также вернуть результат — объект данных, содержащий копию присвоенного значения. Спецификация операции присваивания записывается в форме:

```
присваивание (=) : тип × тип → тип
```

Рассмотрим такое присваивание: `a = a`. Здесь интересно проследить различные интерпретации ссылок на переменную `a`. Стоящее справа `a` ссылается на величину, содержащуюся в переменной с этим именем (ссылка на содержание). Подобные ссылки называются *правосторонними значениями* (операции присваивания) или *r*-значениями объекта данных. В свою очередь, стоящее слева `a` ссылается на местоположение объекта, в котором будет содержаться новое значение (ссылка на местоположение). Такие ссылки называются *левосторонними значениями* (операции присваивания), или *l*-значениями. Используя эти термины, операцию присваивания нужно определить следующим образом:

1. Вычислить *l*-значение первого операнда выражения.
2. Вычислить *r*-значение второго операнда выражения.

3. Присвоить вычисленное *r*-значение вычисленному *l*-значению объекта данных;
4. Возвратить вычисленное *r*-значение как результат выполнения операции.

Таким образом, операция присваивания в языке С возвращает присвоенное *r*-значение объекта данных. В языке С широко используются обе возможности операции присваивания: и как механизма изменения значения объекта данных (через его *l*-значение), и как функции, возвращающей значение (через *r*-значение объекта данных).

Порядок вычисления операндов в выражении

В общем случае операндами выражения могут быть переменные, константы, функции и подвыражения в скобках.

Вычисление значений переменных происходит путем выборки их из памяти. Константы или вычисляются тем же способом, или же их значения уже присутствуют в выражении и не требуют выборки из памяти. Вычисление функции сводится к исполнению операторов ее тела. Вычисление подвыражения в скобках требует, в свою очередь, вычисления содержащихся в нем операндов.

Если ни один из операндов в операторе не имеет побочных эффектов, то порядок вычисления операндов значения не имеет и компилятор может установить его сам, решая задачу оптимизации вычислений. Следовательно, особый интерес представляет именно наличие у операндов побочных эффектов.

Практически наиболее важен *побочный эффект функции*. Он возникает, когда функция изменяет один из своих параметров или глобальную переменную. Глобальная переменная объявляется вне функции, но доступна в ней.

Рассмотрим выражение:

```
n × func(m) + n
```

Прежде чем выполнять умножение, следует извлечь из памяти *r*-значение переменной `n`, *r*-значение переменной `m` и вычислить `func(m)`. Для сложения нужен результат операции умножения и значение `n`. Чтобы сократить время вычислений, целесообразно извлечь значение `n` из памяти только один раз, а затем в ходе вычислений использовать его два раза. Кроме того, вычисление `fun(m)` и извлечение `n` из памяти не должны зависеть друг от друга. Теперь предположим, при вычислении функции из-за побочного эффекта изменяется значение `n`. В этой ситуации точный порядок вычисления приобретает первостепенное значение. Например, если вначале в ячейке памяти для `n` хранилась `1`, а `func(m)` возвращает результат `4` и также изменяет величину `n` на `1`, тогда различные порядки вычислений приводят к разным результатам:

вычисления в порядке слева направо дают результат: `1 × 4 + 2 = 6`.
`n` *вычисляется только один раз, причем до вычисления* `func(m)`: `1 × 4 + 1 = 5`.
значение `func(m)` *вычисляется до вычисления* `n`: `4 × 2 + 2 = 10`.

Во всех этих вариантах вычисления корректны (с точки зрения правил языка), но отличаются порядком вычисления операндов.

Заметим, что в математике функции не имеют побочных эффектов, поскольку здесь нет понятия переменной. То же самое справедливо и для языков функционального программирования. В математике и языках функционального программирования функции гораздо проще для понимания, чем в императивных языках, потому что их контекст никак не связан с их смыслом.

Проблема порядка вычисления операндов и побочных эффектов имеет два возможных решения:

1) не допускать использования в выражениях функций с побочным эффектом;
2) разрешать побочные эффекты, но в определении языка задать точный порядок вычисления операндов в выражениях.

Запретить побочный эффект функций довольно трудно, поскольку это существенно снижает гибкость программирования. Ведь речь идет об исключении целого арсенала функций с двусторонними параметрами и запрете на доступ к глобальным переменным. Отказ от функций, возвращающих несколько значений, резко сужает возможности программистов, а обращение к глобальным данным, вне механизма передачи параметров, считается важным методом увеличения эффективности и скорости выполнения программ. В частности, компиляторы весьма широко применяют глобальный доступ к таблицам символов.

С другой стороны, установка строгого порядка вычислений конфликтует с методами оптимизации программ. В рамках данного подхода из определения языка должен следовать совершенно ясный порядок вычисления операндов, чтобы программист мог *правильно* использовать побочные эффекты. Однако многие оптимизирующие компиляторы изменяют порядок вычислений операндов внутри выражений. Это означает: при строго заданном порядке вычисления определенные виды оптимизации становятся невозможными. Таким образом, лучшего решения просто нет, что и подтверждается существующими языками.

Для операторов возможность побочных эффектов обычно допускается. Самый распространенный оператор присваивания всегда производит побочный эффект, изменяя значение элемента данных. Достаточно часто побочный эффект одного оператора влияет на исходные данные другого оператора.

Особым видом побочного эффекта считают аварийное (неуспешное) завершение функции, приводящее к генерации состояния ошибки. Такой побочный эффект характерен не только для высокоуровневых функций, созданных программистом, но и для элементарных операций (примеры: переполнение, деление на ноль и т. д.). Запрещать, подавлять подобные эффекты, конечно, не принято. Нужно лишь продумывать всевозможные условия их появления и варианты влияния на ход вычислений.

Очень часто для организации ветвления применяют сложные условия, состоящие из набора простых выражений отношения. В роли «клея», сочленяющего выражения отношения, в них выступают логические (булевы) операции `И` (`&&` в языке C) и `ИЛИ` (`||` в языке C). В языке C выполнение этих операций соответствует короткой схеме вычислений. По этой схеме второй операнд логической операции вычисляется не всегда. В частности, справедливо следующее:

- значением булева выражения `a || b` будет *истина*, если значение `a` — *истина*;
- значением булева выражения `a && b` будет *ложь*, если значение `a` — *ложь*.

В данных случаях нет нужды вычислять значение операнда b.

Приведем два примера, записанных на языке C:

```
(A == 0) || (B/A > C)
```

и

```
(I <= MAX) && (Z[I] > N)
```

Смысл этих выражений состоит в следующем: вычисление правого операнда булевой операции может привести к побочному эффекту — возникновению ошибки, а именно:

- деление на ноль — для верхнего примера;
- выход индекса за пределы допустимого диапазона — для нижнего примера.

Левый операнд призван предотвратить такую ошибку.

Правый операнд не вычисляется:

- если левый операнд (A == 0) получил значение *истина*;
- если левый операнд (I <= MAX) получил значение *ложь*.

Увы, но на уровне реальной реализации большинства языков эта идея не срабатывает, поскольку оба операнда вычисляются еще до сканирования символа булевой операции. Многие ошибки возникают из-за неоправданного расчета на то, что значение левого операнда булевой операции может отменить вычисление ее правого операнда. Самая надежная защита от подобных ошибок предложена в языке Ada, где в дополнение к обычным булевым операциям and и or, которые не вычисляются по короткой схеме, введены две специальные булевы операции, and then и or else, явным образом осуществляющие вычисление по короткой схеме. Например, на языке Ada выражение

```
(A = 0) or else (B/A > C)
```

не может сгенерировать ошибку деления на ноль, так как при A = 0 вычисление всего выражения прекращается и его значение считается истинным.

Контрольные вопросы и упражнения

1. Переписать следующие выражения в префиксной нотации. Функцию sqrt представить как операцию с одним аргументом:

```
а) a * b + c
б) a * (b + c)
в) a * b + c * d
г) a * (b + c) * d
д) [b / 2 + sqrt((b / 2) * (b / 2) – a * c)] / a
```

Решение:

```
а) + * a b c
б) * a + b c
в) + * a b * c d
г) * * a + b c d
д) / + / b 2 sqrt – * / b 2 / b 2 * a c a
```

2. Переписать выражения из упражнения 1 в постфиксной нотации.

   Решение:

   ```
   а) a b * c +
   б) a b c + *
   в) a b * c d * +
   г) a b c + * d *
   д) b 2 / b 2 / b 2 / * a c * - sqrt + a /
   ```

3. Вычислить значение постфиксного выражения «`7 7 * 4 2 * 3 * - `» с помощью структуры данных `стек`. Стек следует отображать в виде прямоугольника, растущего вниз. Стек на каждом шаге вычисления должен содержать конкретные элементы. Вершина стека на всех шагах вычисления должна находиться на одном и том же уровне. Направление перемещения элементов (чисел и/или значков операции) нужно показать стрелкой, расположенной над вершиной стека.

Глава 5

Действия и операторы в программах

В этой главе представляется весь спектр операторов, применяемых в языках программирования: операторы для создания линейных, разветвляющихся и циклических программ. Обсуждаются простые операторы и операторы перехода, после чего вводится понятие потока управления в программе и все остальные средства рассматриваются с позиций обеспечения его структурированности. Для удобства изложения операторы разветвления разделены на условные операторы (ветвление на два направления) и операторы выбора (ветвление на много направлений), а операторы циклов классифицируются по основанию «число повторений». Среди циклов с заданным числом повторений отмечены циклы с повторением по данным, достаточно популярные в настоящее время. В свою очередь, среди операторов с не заданным числом повторений выделяются бесконечные циклы, циклы с предусловием и циклы с постусловием. Все операторы иллюстрируются многочисленными примерами из популярных императивных, скриптовых, объектно-ориентированных и функциональных языков. В качестве поучительного примера средств управления с недетерминированной семантикой описываются охраняемые структуры Дейкстры. В заключение поясняется идея инвариантов и их использование при написании программ.

Базовые операторы

Удивительно, но основными «рабочими лошадками» в императивных программах являются выражения и операторы присваивания: выражения определяют порядок вычисления значений, а операторы присваивания обеспечивают занесение значений в некоторые переменные.

Суть операции присваивания обсуждалась в предыдущей главе. Основным назначением присваивания считается присвоение *l*-значению объекта данных (то есть области памяти, выделенной для объекта) *r*-значения (то есть значения объекта данных) определенного выражения. Синтаксис представления явного присваивания от языка к языку существенно меняется (табл. 5.1).

В семействе языков, сходных по синтаксису с языком C, присваивание считается просто операцией, поэтому можно написать: `y = x = 5`. Поскольку присваивание

является правоассоциативной операцией, данная запись означает `y = (x = 5)`, и ей соответствует следующий порядок вычислений:

1. Переменной `x` присваивается значение `5`.
2. Выражение `(x = 5)` возвращает значение `5`.
3. Переменной `y` присваивается значение `5`.

Таблица 5.1. Обозначения оператора присваивания в разных языках программирования

Обозначение	Языки
`Y := X`	Pascal, Ada
`Y = X`	C, C++, Java, Visual Basic, FORTRAN, PL/1, Prolog, ML
`MOVE X TO Y`	COBOL
`Y ← X`	APL
`(SETQ Y X)`	LISP

И все же чаще всего присваивание используется только в качестве отдельного оператора. Примером может служить оператор присваивания в языке Pascal:

```
Y := 2 + Z;
A := 4 + 3 * B;
```

Другие возможности здесь исключены, так как операция присваивания языка Pascal не возвращает никакого явного результата

В большинстве языков присваивания применяются лишь отдельно. Однако, как показано в табл. 5.2, в языке C их комбинируют с другими арифметическими операциями.

Таблица 5.2. Комбинированные операции присваивания в языке C

Синтаксис	Описание
`Y = X`	Присваивает *r*-значение переменной X *l*-значению переменной Y, возвращает *r*-значение переменной Y
`Y += X`	Увеличивает значение переменной Y на значение переменной X (`Y = Y + X`), возвращает новое значение
`Y -= X`	Уменьшает значение переменной Y на значение переменной X (`Y = Y - X`), возвращает новое значение
`Y *= X`	Умножает значение переменной Y на значение переменной X (`Y = Y * X`), возвращает новое значение
`Y /= X`	Делит значение переменной Y на значение переменной X (`Y = Y / X`), возвращает новое значение

Такие комбинированные операции присваивания позволяют сократить запись присваиваемых выражений.

Очень интересные возможности дает применение в операторах присваивания для языка C операций изменения значения переменных на единицу. Речь идет об операции увеличения значения (`++`) и операции уменьшения значения (`--`). Первая называется операцией инкремента, вторая — операцией декремента.

Возможна префиксная и постфиксная формы инкремента и декремента:

- Если операции инкремента и декремента записаны в префиксной форме, то переменная сначала изменяется, а затем ее новое значение используется в выражении.
- Если операции инкремента и декремента записаны в постфиксной форме, то в выражении используется текущее значение переменной, и только после этого ее значение изменяется.

Использование этих форм иллюстрирует табл. 5.3.

Таблица 5.3. Префиксная и постфиксная формы инкремента и декремента

Синтаксис	Описание
`++A`	Увеличивает значение переменной A на 1 и возвращает новое значение (`A = A + 1`), возвращает *r*-значение A
`--A`	Уменьшает значение переменной A на 1 и возвращает новое значение (`A = A - 1`), возвращает *r*-значение A
`A++`	Возвращает значение переменной A, затем увеличивает ее значение на 1 (сначала возвращает *r*-значение A, затем `A = A + 1`)
`A--`	Возвращает значение переменной A, затем уменьшает ее значение на 1 (сначала возвращает *r*-значение A, затем `A = A - 1`)

ПРИМЕЧАНИЕ

Если операции инкремента и декремента осуществляются в виде отдельного оператора, то обе формы приводят к одинаковому результату.

Большинство языков программирования содержат операторы для чтения вводимых пользователем данных с клавиатуры, из каналов связи или из файлов. Эти операторы также изменяют значение переменных через операции присваивания. Обычно синтаксис таких операторов выглядит как: `read(file, data)`. В языке C вызов оператора `scanf` приводит к присваиванию переменной некоторого значения из стандартного файла ввода. В языке Perl операцию чтения вызывает упоминание стандартного входного файла. Например, `$A = <STDIN>` вызывает присваивание переменной `$A` очередной строки ввода.

Конечно, можно задать линейные вычисления, записав один за другим последовательность операторов присваивания. И все же количество полезных программ, состоящих исключительно из операторов присваивания, крайне ограничено. Чтобы сделать программы более гибкими и разнообразными, необходимы средства для организации разветвлений и повторного выполнения некоторых операторов. Операторы, предоставляющие такие возможности, обычно называют операторами управления.

Каким должен быть простой и универсальный набор операторов управления? Простой в том смысле, чтобы любой оператор можно было без труда освоить и легко применить при создании программы. Универсальность же набора должна покрывать все мыслимые и немыслимые программные приложения. Ответ на этот вопрос широко дискутировался на протяжении последних двадцати пяти лет XX века [9–12, 17, 19]. По сути, речь шла о разумном расширении состава операторов языка программирования, эффективно увеличивающего легкость программирования за счет баланса простоты, универсальности и читабельности.

Как правило, оператор управления является структурой, в состав которой входит одна или несколько последовательностей операторов присваивания. Исследования 60-х годов XX века доказали, что структуры управления должны иметь один вход и один выход. Наличие нескольких входов в таких структурах делает программы менее читабельными и понятными.

Операторы перехода

Оператор перехода является простейшим средством изменения линейного порядка выполнения базовых операторов, он передает управление выполнением в любое указанное место программы.

Передача управления чаще всего указывается с помощью `goto` — оператора перехода на оператор с заданной меткой.

В языках программирования возможны два вида оператора `goto`:

- *Безусловный* `goto`. Если в последовательности операторов встречается оператор безусловного перехода `goto`, например:

  ```
  goto NEXT_OP
  ```

  то управление передается оператору, помеченному меткой `NEXT_OP`, а оператор, непосредственно следующий за оператором `goto`, не выполняется.

- *Условный* `goto`. Если в последовательности операторов встречается оператор условного перехода `goto`, например:

  ```
  if X > Y then goto NEXT_OP
  ```

  то управление передается оператору, помеченному меткой `NEXT_OP`, только в случае выполнения заданного условия `X > Y`.

Внешний вид разрешаемых меток меняется от языка к языку. В языке C для меток используют синтаксис идентификаторов, в языках FORTRAN и Pascal — беззнаковые целые константы.

Уместно заметить, за последние 40 лет оператору `goto` пришлось пережить очень трудные времена. Его использование при написании программ подвергалось суровой критике.

Конечно, оператор `goto` является самым мощным и простым оператором управления потоком вычислений. Однако, перефразируя знаменитое высказывание М. Жванецкого, одно неосторожное использование такого оператора — и ты отец... проблемы. Он демонстрирует ошеломляющую мощь и гибкость. Достаточно сказать, что все другие структуры управления легко создаются с помощью оператора `goto`, но эта сила, увы, и определяет его предельную опасность. Применение `goto` без ясных ограничений может существенно понизить читабельность, надежность и простоту сопровождения программы.

Проблемы являются следствием способности оператора `goto` выполнять головокружительные «прыжки» через фрагменты программы. А ведь известно, что проще всего читается (и понимается) программа, операторы которой выполняются в порядке их размещения, то есть в порядке «сверху-вниз». Частичным решением проблемы может стать такое ограничение: оператору `goto` разрешают передавать

управление в программе только вниз. Впрочем, в аварийных ситуациях ограничение может сниматься.

Впервые разъяснил опасность операторов `goto` Э. Дейкстра [60], который призывал чуть ли не сжечь его на костре и т. д. Среди сторонников более умеренных взглядов назовем известного ученого Д. Кнута, предоставившего примеры, в которых эффективность оператора `goto` перевешивала его вред для читабельности [71].

Было создано несколько языков программирования без оператора `goto` (например, Java). Однако самые современные языки программирования включают операторы перехода. Был он включен и в язык С, несмотря на заявление его автора о «бесконечной неправильности» оператора перехода. В языках программирования без оператора `goto` предусматриваются дополнительные операторы управления, компенсирующие его отсутствие. Как правило, это операторы цикла или выхода из подпрограмм.

В языке С в качестве формы более безопасного управления переходами предусмотрен оператор `break`. Этот оператор может размещаться внутри операторов управления `while`, `for`, `switch`. Оператор `break` передает управление оператору, непосредственно следующему за структурой управления, в которой он сам содержится. Считается, что использование оператора `break` не ухудшает читабельность программы.

Помимо этого, язык С имеет родственный оператору `break` оператор `continue`. Забегая вперед, отметим: данный оператор завершает выполнение текущей итерации цикла в операторах `while` и `for` и переводит управление в конец тела цикла. Таким образом, оператор `break` прерывает цикл, в то время как оператор `continue` досрочно передает управление на следующую итерацию цикла.

Иногда неправильное размещение оператора `break` внутри оператора управления `switch` нарушает стабильность вычисления программ на языке С. Снова забегая вперед, отметим, если нет оператора `break`, то от одной ветви оператора `switch` управление переходит к следующей. Может возникнуть путаница, из какой именно ветви (при наличии оператора `break`) следует выйти.

Поток управления

Вернемся к обсуждению потока управления в программе. Программа выполняется компьютером в автоматическом режиме. При выполнении программы происходит передача управления отдельным операторам. Каждый оператор в течение кванта времени управляет компьютером. Говорят, что вычисления задает поток управления в программе.

Поток управления — это последовательность передач управления между операторами. Он задает порядок выполнения операторов.

Программа вводится в компьютер как последовательность строк. Отдельные строки программы могут выполняться неоднократно, приводят к долгим вычислениям. Например, статическая строка

```
for i from 1 to 3 do writeln (i, i * i );
```

вызывает динамические вычисления:

```
writeln (1, 1 * 1)
writeln (2, 2 * 2)
writeln (3, 3 * 3)
```

Статический текст программы может значительно отличаться от динамических вычислений, выполняемых при работе программы. Это затрудняет понимание программы, увеличивает возможность внесения ошибки.

Для облегчения понимания операторы императивного языка программирования должны обеспечивать *структурированный поток управления*.

Структурированным называют поток управления, который удобен для понимания человеком и максимально близок к структуре текста программы.

Программа является *структурированной*, если ее поток управления очевиден из структуры программного текста.

Многочисленные исследования показали, что структурированность обеспечивают языки, операторы которых имеют по одному входу и одному выходу. Структурированный поток управления может принимать линейную, разветвляющуюся и циклическую формы (рис. 5.1). Считается, что эти формы позволяют человеку легко отслеживать логику вычислений в программе. На рисунке потоку управления соответствует последовательность стрелок, отображающих управляющие связи между операторными вершинами.

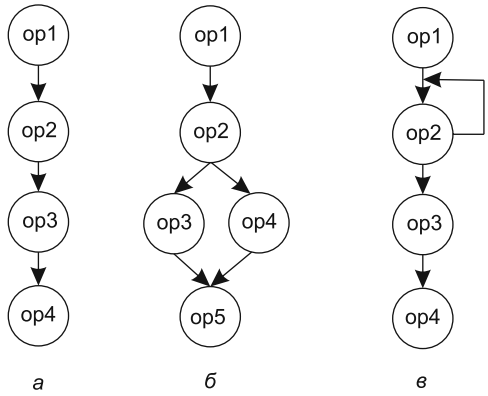


Рис. 5.1. Формы структурированного потока управления:
а — линейная; *б* — разветвляющаяся; *в* — циклическая

Средствами обеспечения структурирования являются три структуры управления: композиция, ветвление, цикл [73]:

- *Композиция*. Эта структура является средством группировки линейной последовательности простых операторов.
- *Ветвление*. Структура обеспечивает выбор для выполнения одной из нескольких альтернативных последовательностей операторов.
- *Цикл*. Последовательность операторов может повторяться многократно, ноль или более раз. Ноль означает, что последовательность может быть вообще пропущена.

Эти структуры рассматриваются как строительные блоки, из которых, многократно комбинируя их, мы можем создать сколь угодно большие структурированные программы. Процесс же конструирования программ из этих блоков называют структурированным (а иногда — структурным) программированием.

Здесь возникает законный вопрос: не приведет ли использование только таких структур управления к ограничению возможностей программ и программирования как такового? Ответ давался многими исследователями, утверждавшими: ограничения на общность и универсальность решений не возникают! Среди наиболее ярких результатов отметим структурную теорему Бема–Джакопини [53] и теорему о регуляризации алгоритмов выдающегося отечественного ученого В. М. Глушкова [6].

Впрочем, надо иметь в виду следующее: фактически доказано лишь то, что любую программу *можно* написать, используя только перечисленные структуры управления. Получаемый результат вовсе не обеспечивает автоматически лучшее решение. Известно, что существует некоторое решение *не хуже* (а может, и лучше) структурированного результата.

В языках программирования описанные структуры управления реализуются специальным набором операторов управления, которые представляют собой операторы с одной точкой входа и одной точкой выхода [9–12, 17]. Каждый оператор управления может содержать внутренние ветвления и циклы, но передача управления из него к другому оператору происходит только через его единственную точку выхода. Все эти рассуждения ориентированы на поддержку следующего утверждения: порядок выполнения операторов программы должен соответствовать порядку операторов в ее тексте.

Составные операторы

Составной оператор представляет собой последовательность операторов, которую при создании более сложных операторов можно рассматривать как единый оператор. Впервые составной оператор появился в языке Algol 60. Обычно составные операторы записывают в виде:

```
begin
…          — последовательность из нескольких операторов
end
```

Здесь несколько операторов помещаются между открывающей `begin` и закрывающей `end` скобками. Пример составного оператора:

```
begin
    temp := x;
    x := y;
    y := temp
end
```

ПРИМЕЧАНИЕ

В Паскале между операторами последовательности ставят точки с запятой. Лучший вариант — точки с запятой должны завершать операторы, то есть появляться после каждого оператора.

В языках C, C++, C#, Perl и Java границы составного оператора обозначаются открывающей и закрывающей фигурной скобкой: {…}. В некоторых языках (C, Ada) в начало составного оператора можно добавлять *объявления данных*, превращая его в *блок*.

Операторы, входящие в составной оператор, записываются в той последовательности, в которой они должны выполняться. Таким образом, составной оператор группирует линейный поток управления и реализует структуру управления «композиция».

Поскольку составной оператор — это отдельный оператор, то операторы некоторого фрагмента вычисления можно объединять вместе, применяя скобки `begin … end`, причем можно составлять иерархические структуры из таких фрагментов.

Язык Pascal унаследовал от языка Algol способ формирования составных операторов, но не позволяет создавать блоки. Некоторые языки не нуждаются в специальном разграничении составных операторов, интегрируя их в свои структуры управления.

Условные операторы

Условный оператор обеспечивает возможность альтернативного выполнения одной из двух последовательностей (ветвление) или выполнение какой-либо одной последовательности при определенных условиях. Термин *последовательность* здесь обозначает либо *одиночный* элементарный оператор, либо составной оператор, либо другой оператор управления. Выбор осуществляется путем проверки *выражения ветвления*, результат которой имеет истинное или ложное значение.

В самом общем виде можно записать условный оператор с двумя ветвями:

```
if выражение_ветвления  then последовательность_1
                        else последовательность_2;
```

В зависимости от вычисленного значения выражения ветвления (*истина* или *ложь*), выполняется соответственно либо `последовательность_1` операторов, либо `последовательность_2` операторов.

Условный оператор с одной ветвью

```
if выражение_ветвления  then последовательность;
```

обеспечивает выполнение последовательности при значении условия *истина* и пропуск последовательности при значении *ложь*.

Приведем пример условного оператора на языке Pascal:

```
if x > 0     then y := 1/x
             else
             begin
                  x := 2;
                  y := 1/z
             end
```

Последовательность в ветви `then` здесь является простым оператором присваивания, а последовательность в ветви `else` — составным оператором. Заметим, что точку с запятой в конце then-последовательности ставить нельзя.

Перепишем этот оператор для языка C:

```
if (x > 0) y = 1/x;
    else {
        x = 2;
        y = 1/z;
    }
```

В этом языке (и языках C++, C#, Java) нет ключевого слова `then`, а проверяемые выражения в условных операторах и операторах цикла должны заключаться в круглые скобки. Выражение ветвления здесь считается арифметическим выражением, которое возвращает значение 0 (ноль моделирует *ложь*) или 1 (единица моделирует *истину*). В языках C++, C#, Java для этих целей используют логические значения. Наконец, точку с запятой в конце then-последовательности здесь надо ставить обязательно.

Арифметическое выражение в ветвлении используется также в языке Python, а если быть точными, то в C++ может применяться как арифметическое, так и логическое выражение.

Во многих современных языках ветви `then` и `else` могут содержать как единичные, так и составные операторы. Исключением является язык Perl, в котором обе ветви должны быть составными операторами, даже если они содержат единичные операторы. Многие языки для формирования составных операторов используют скобки, которые ограничивают «тела» ветвей `then` и `else`. В языках Fortran 95, Ada, Python и Ruby ветви `then` и `else` являются последовательностями операторов, а не составными операторами. Весь условный оператор завершается в этих языках зарезервированным словом или фразой (например, в языке Ada применяется `end if`).

Язык Python для спецификации составных операторов использует отступы. Приведем пример:

```
if a > b :
    a = b
    print "первый вариант"
```

Все операторы с одинаковым отступом включены в составной оператор. Оператор, следующий за составным оператором, должен иметь такой же отступ, что и предложение `if`. Заметим, что вместо `then` здесь используется двоеточие.

В функциональных языках ML, F# и LISP разветвление не является оператором: это выражение, которое возвращает значение. Следовательно, оно может появиться там, где уместны любые другие выражения. Рассмотрим следующий пример на языке F#:

```
let m =
    if a > 0 then a
    else 5 * a;;
```

Здесь создается имя `m`, которое инициируется величиной `a` или выражением `5 * a`, в зависимости от результатов анализа значения `a`.

Вариации формы ветвей влияют на смысл вложенных условных операторов.

Вложенность условных операторов

Интересная проблема появляется, если допускается вложение условных операторов. Рассмотрим следующий условный оператор на языке Pascal:

```
if x1 > y1 then
    if x2 > y2 then
    оператор_1
else
    оператор_2;
```

Здесь возникает вопрос — какому из двух `if` соответствует `else`? Отступы в написанном тексте намекают на желание программиста связать `else` с первым `if`. Но это лишь желание... Увы, синтаксис не дает указаний о соответствии. За исключением языков Python и F#, отступы не влияют на семантику и игнорируются компиляторами. Суть проблемы состоит в том, что ветвь `else` следует за двумя ветвями `then` без промежуточной ветви `else`, и нет никаких синтаксических индикаторов, указывающих на соответствие ветви `else` одной из ветвей `then`.

Для предотвращения неоднозначности в языках Pascal, Java, как и во многих других императивных языках, придумывается дополнительное, внутреннее правило, связывающее `else` с ближайшим предшествующим `if` и, следовательно, со вторым `then`. Говорят, что ветвь `else` всегда образует пару с ближайшей предшествующей ветвью `then`, не имеющей пары. Следовательно, в этом примере ветвь `else` должна составить пару второй ветви `then`. Недостаток использования такого правила состоит в том, что, хотя программист и может считать `else` альтернативой первому `then`, компилятор на уровне синтаксиса его ошибку не выявит, но будет считать семантику прямо противоположной. Чтобы ввести (например, на языке Java) альтернативную семантику, нужно внутренний `if` сделать составным оператором:

```
if (x1 > y1) {
    if (x2 > y2)
    оператор_1;
}
else
    оператор_2;
```

В языках C, C++ и C# существует та же проблема вложенности условного оператора. Поскольку язык Perl требует, чтобы все ветви `then` и `else` были составными, здесь проблемы вложенности нет. На языке Perl предыдущий пример записывается в следующем виде:

```
if (x1 > y1) {
    if () {
    оператор_1;
    }
} else {
    оператор_2;
}
```

Если же требуется альтернативная семантика, то надо записать:

```
if (x1 > y1) {
    if (x2 > y2) {
        оператор_1;
    }
    else {
        оператор_2;
    }
}
```

Другое решение проблемы вложенности условного оператора на уровне синтаксиса, заключается в формировании составных операторов иными средствами. Рассмотрим синтаксическую структуру условного оператора в Java. Ветвь `then` следует за выражением ветвления, а ветвь `else` начинается с зарезервированного слова `else`. Когда ветвь `then` содержит единичный оператор и присутствует ветвь `else`, зарезервированное слово `else` фактически отмечает границу ветви `then` (хотя в этом и нет необходимости). Когда ветвь же `then` является составным оператором, она завершается закрывающей фигурной скобкой. Однако для условного оператора с замыкающей ветвью (будь то `then` или `else`), содержащей простой оператор, *синтаксическая* закрывающая скобка не предусмотрена. Использование для этой цели специального слова устраняет трудности, связанные с семантикой вложения условных операторов, и увеличивает надежность ветвления. Такое проектное решение для условного оператора реализовано в языках Fortran 95, Ada, Ruby и Lua. К примеру, рассмотрим следующий оператор на языке Ruby:

```
if x > y then
    sum = sum + x
    rez = rez + 1
else
    sum = sum + y
    rez = rez - 1
end
```

Этот оператор удобнее условных операторов в C-подобных языках, поскольку его форма не зависит от количества операторов в ветвях `then` и `else` (это же справедливо для Perl). В языке Ruby ветви `then` и `else` состоят из последовательности операторов, а не из составных операторов. Исходный пример этого раздела на Ruby имеет следующий вид:

```
if x1 > y1 then
    if x2 > y2 then
        оператор_1
    else
        оператор_2
    end
end
```

Теперь тайное стало явным и нет никакой неопределенности. Такая конструкция имеет более четкую структуру, чем конструкция ветвления в языке Pascal, C-подобных языках, так как ее форма не зависит от количества внутренних условных операторов. Поскольку вложенный if-оператор закрывает зарезервированное слово `end`, становится совершенно ясно, что ветвь `else` является парой внутренней ветви `then`.

Вторая интерпретация начального примера, в которой ветвь `else` должна относиться к внешнему `if`, на Ruby записывается в следующей форме:

```
if x1 > y1 then
    if x2 > y2 then
        оператор_1
    end
else
    оператор_2
end
```

Следующий оператор, представленный на языке Python, семантически эквивалентен последнему оператору, записанному на Ruby:

```
if x1 > y1 :
    if x2 > y2 :
        оператор_1
else:
    оператор_2
```

Если бы строка `else:` имела такой же отступ, который был принят для вложенного `if`, то ветвь `else` превращалась бы в пару для внутреннего `if`.

Разработчики языка Ada для условного оператора ввели закрывающую скобку — ключевое слово `end if`. Запись исходного примера на языке Ada примет вид:

```
if x1 > y1 then
    if x2 > y2 then
        оператор_1;
    else
        оператор_2;
    end if;
end if;
```

Здесь тоже все ясно и нет никакой неопределенности. Если ключевое слово `end if` закрывает вложенный оператор `if`, значит `else` соответствует внутреннему `then`.

Кстати, теперь нет нужды помещать ветви, образуемые длинными последовательностями операторов, в скобки составных операторов.

Правда, появляется побочный эффект — размножение закрывающих скобок.

Например, при воспроизведении конструкции языка Pascal

```
if E1 then оп-р1
    else if E2 then оп-р2
    else if E3 then оп-р3
    else оп-р4
```

на языке Ada получаем:

```
if E1 then оп-р1
    else if E2 then оп-р2
        else if E3 then оп-р3
            else
                оп-р4
            end if;
        end if;
end if;
```

Количество закрывающих скобок сокращается при введении вложенного оператора ветвления `elsif` (у которого закрывающая скобка отбрасывается за ненадобностью):

```
if  E1  then  оп-р1
elsif  E2  then  оп-р2
elsif  E3  then  оп-р3
else
    оп-р4
end  if;
```

Конструкции `else-if` основываются на общей математической конструкции — условном выражении, они позволяют лаконично и прозрачно описать ветвление на много направлений. Подобные конструкции применяются и в других языках, например в языках Perl и Python. Рассмотрим следующий условный оператор

на языке Python (здесь вложенные операторы начинаются с зарезервированного слова `elif`):

```
if weight < 61 :
    lightweight = True
elif weight < 72 :
    middleweight = True
elif weight < 90 :
    cruiserweight = True
else :
    heavyweight = True
```

Этот оператор эквивалентен следующему вложению условных операторов:

```
if weight < 61 :
    lightweight = True
else :
    if weight < 72 :
        middleweight = True
    else :
        if weight < 90 :
            cruiserweight = True
        else :
            heavyweight = True
```

Ясно, что `else-if` версия более лаконична и читабельна.

У языка ML нет проблемы вложенности условных операторов, поскольку здесь в условной конструкции не предусмотрена ветвь `else`.

Операторы выбора

Оператор выбора позволяет произвести ветвление по одному из многих вариантов. Причем каждый вариант может включать один оператор или целую группу операторов. Следовательно, оператор выбора может рассматриваться как обобщение условного оператора. Конечно, ветвление по многим направлениям можно реализовать и с помощью условных операторов, но такое решение оказывается трудным для чтения, громоздким и ненадежным.

Оператор выбора в языке Pascal называется case-оператором и имеет следующий вид:

```
case выражение of
    список констант 1: оператор_1;
    список констант n: оператор_n;
end
```

Количество значений выражения должно быть счетным (дискретным). Выбираемые операторы могут быть либо отдельными, либо составными. Семантика оператора заключается в следующем: вычисляется выражение, и его значение сравнивается с константами во всех списках констант. Если найдено совпадение, то управление передается оператору, приписанному к соответствующей константе. После выполнения этого оператора происходит выход из case-оператора.

Константы в разных списках должны быть различными, то есть константы не могут появляться в нескольких списках констант. Требование о покрытии константами всех значений выражения не выдвигается. Эта проблема была просто проигнорирована. Во многих современных версиях этого языка в case-оператор

добавляется ветвь `else`, которая выполняется, если значение выражения не принадлежит ни одному из списков констант.

Формы оператора выбора в различных языках различны, однако все они гарантируют следующее:

- константы вариантов могут появляться в любом порядке;
- с помощью различных констант может выбираться один и тот же вариант;
- константы разных вариантов должны быть различны.

Оператор выбора классической версии Pascal был улучшен в языке Ada: добавлена альтернативная часть, в список констант введены диапазоны. Позднее эти нововведения попали в Turbo Pascal.

Оператор выбора в языке Ada позволяет сделать выбор из произвольного количества вариантов, имеет вид:

```
case  выражение is
    when  список выбора 1 =>
        последовательность операторов 1;
    ...
    when  список выбора n  =>
        последовательность операторов n;
    when others =>
        последовательность операторов n+1;
end case;
```

Порядок выполнения оператора:

1. Вычисляется значение выражения.
2. Каждый список выбора (от первого до последнего) проверяется на соответствие значению.
3. Если найдено соответствие, то выполняется соответствующая последовательность операторов, после чего происходит выход из оператора `case`.
4. Если не найдено соответствие, то выполняются операторы после условия `when others`.

Элементы списка выбора отделяются друг от друга вертикальной чертой (' | ') и могут иметь вид:

- `выражение`.
- `выражение n .. выражение m`.

Примеры

```
a)    case Number is
        when 1 | 7 => Put ("Is 1 or 7");
        when 5 => Put ("Is 5");
        when 25..100 => Put ("Is number between 25 and 100");
        when others => Put ("Is unknown number");
    end case;
б)    case Answer is
        when 'A'..'Z' | 'a'..'z' => Put_Line ("It's a letter!");
        when others => Put_Line ("It's not a letter!");
    end case;
```

Оператор выбора `switch` в языке C, входящий также и в языки C++ и Java, достаточно прост. Структура `switch`-оператора представляется в следующем виде:

```
switch (выражение) {
    case выражение_константа_1: оператор_1; break;
    …
    case выражение_константа n: оператор_n; break;
    [default: оператор_n+1]
}
```

Здесь переключающее выражение и выражения-константы должны иметь целые или символьные значения.

Операторы вариантов могут быть последовательностями операторов, составными операторами или блоками, но чаще всего должны заканчиваться оператором `break`. Оператор `break`, как уже упоминалось ранее, играет роль ограниченного `goto`. Его выполнение приводит к выходу из оператора выбора. Если же программист по ошибке пропустит в `switch`-операторе оператор `break`, то управление передается на операторы следующего варианта. Это может быть ошибкой, хотя создатели языка сознательно приняли такое проектное решение. Они считали, что повышают гибкость программирования (появляется возможность с помощью одной константы «запустить» несколько вариантов ветвления). На практике, однако, такая возможность используется достаточно редко.

Если переключающее выражение принимает значение, отличное от предусмотренных констант, то выполняется оператор `n+1 default`-варианта. Вариант `default` в операторе `switch` может отсутствовать. Если ни одна константа не совпала со значением выражения и вариант `default` отсутствует, то не выполняются никакие действия.

Рассмотрим пример использования `switch`-оператора в языке C:

```
switch (x - 1){
    case 0:
        y = 0;
        z = 2;
        break;
    case 2:
    case 3:
    case 4:
    case 5:
        y = 3;
        z = 1;
        break;
    case 7:
    case 9:
        z = 10;
        break;
    default:
        /* ничего не делать */
        break;
}
```

Заметим, что здесь сознательно опущены операторы `break` в вариантах с константными значениями 2–4. Все они обслуживаются последовательностью операторов, вписанных в вариант со значением 5. То же самое можно сказать о варианте 7.

Язык Ada позволяет показать классическое решение этой задачи:

```
case x – 1 is
    when 0 =>
```

продолжение ⇩

```
        y := 0;
        z := 2;
    when 2 .. 5 =>
        y := 3;
        z := 1;
    when 7 | 9 =>
        z := 10;
    when others =>
        null;
end case;
```

Оператор `switch` языка C# отличается от оператора выбора для языка C по двум пунктам. Во-первых, в языке C# запрещается неявное выполнение более одного варианта ветвления. Каждый вариант должен заканчиваться явным оператором безусловного перехода: либо оператором `break`, либо оператором `goto`, который может передать управление одному из имеющихся вариантов (или куда-нибудь еще). Во-вторых, переключающее выражение и константы могут быть строками языка C#.

Приведем пример, где число классифицируется по знаку и значению:

```
switch (number) {
    case -1:
        Negatives++;
        break;
    case 0:
        Zeros++;
        goto case 1;
    case 1:
        Positives++;
    default:
        Console.WriteLine("Error in switch \n");
}
```

Здесь метод `Console.WriteLine` отображает строки C#.

Оператор `switch` в языке PHP заимствует синтаксис аналогичного оператора языка C, но более гибок по отношению к переключающему выражению. Значениями вариантов могут быть строки, обычные целые числа и числа двойной точности. Как и в языке C, здесь не обязателен `break` в конце варианта, выполнение может быть продолжено следующим вариантом.

В языке Ruby многовариантный выбор выполняется с помощью `case`-выражений. Предусмотрены две формы `case`-выражений. Первая форма `case`-выражения семантически эквивалентна списку вложенных условных операторов:

```
case( day )
    when day == 1 then puts("Это понедельник" )
    when day == 2 then puts("Это вторник" )
    when day == 3 then puts("Это среда" )
    when day == 4 then puts("Это четверг" )
    when day == 5 then puts("Это пятница" )
    when day == 6 || day == 7 then puts( "Ура! Выходные!" )
    else puts( "Такого дня не бывает!" )
end
```

Смысл этого `case`-выражения достаточно прозрачен: последовательно, проходом сверху вниз, вычисляются булевы выражения. Вычисления заканчиваются, когда будет получено истинное значение. Выполняется оператор соответствующего

when-варианта, расположенный за словом then. Если вычисления успеха не имели, инициируются действия варианта else.

Вторая форма case-выражения позволяет выполнить многовариантный выбор значения, присваиваемого какой-то переменной:

```
salary = 2000000
season = 'лето'
happy = case
        when salary > 1000000 && season == 'лето' then
                puts( "Все просто здорово!" )
                'Очень счастлив'
        when salary > 500000 && season == 'весна' then
                'Совсем неплохо'
        else
                puts( 'Ужасно' )
                'Несчастлив'
        end
```

В этом примере с помощью case-выражения формируется значение переменной happy. Диапазон возможных значений: 'Очень счастлив', 'Совсем неплохо', 'Несчастлив'. В языках Perl, Python и Lua операторы многовариантного выбора отсутствуют.

В языке Scheme выбор обеспечивается специальной формой функции COND, которая является обобщенной версией математического условного выражения. Язык F# включает выражение match, использующее паттерн совпадения как селектор для обеспечения конструкции многовариантного выбора.

Организация повторения операторов

Повторение некоторой последовательности операторов — это мощное средство для сокращения длины программ и повышения эффективности вычислений. Отсутствие возможности повторения группы операторов привело бы к существенному усложнению программ и резкому возрастанию времени на программирование.

Известны две схемы организации повторений: рекурсия и итерация.

Рекурсия — это такой способ организации повторений, при котором последовательность операторов вызывает сама себя непосредственно либо с помощью других последовательностей. Она будет обсуждаться в главах 13 и 14. Рекурсия является естественной математической формой записи повторяющихся вычислений (на основе функций). Рекурсия считается главным средством поддержки повторений в функциональных языках, которые рассматриваются в главе 20.

Итерация — это способ многократного повторения последовательности операторов, реализуемый в виде цикла. В языках программирования циклы поддерживаются итеративными конструкциями, называемыми операторами циклов.

ПРИМЕЧАНИЕ

В общем случае термин «итерация» имеет два значения: 1) процесс выполнения последовательности повторяемых шагов; 2) отдельный шаг из последовательности повторяемых шагов. Опираясь на второе значение, говорят об «отдельной итерации цикла».

Оператор цикла обеспечивает многократное выполнение встроенных в него операторов, являясь, по сути, структурой управления. В состав оператора цикла входят секция управления и тело.

Тело цикла образует произвольный набор операторов, выполнением которых руководит оператор цикла. Этот набор определяет функциональные возможности цикла. *Секция управления* управляет количеством повторений тела.

К настоящему времени разработано несколько разновидностей операторов цикла. Их специфика определяется механизмом работы секции управления и ее размещением относительно тела цикла.

Секция управления может быть размещена как перед телом, так и после тела цикла.

В основу механизма работы секции управления может быть положен анализ логического выражения или подсчет числа повторений. Иногда эти две возможности комбинируют. В результате формируется *условие продолжения*, или *условие завершения цикла*.

Кроме того, различают операторы цикла *с предусловием* (проверка условия предваряет выполнение тела) и *с постусловием* (проверка условия проводится после выполнения тела).

Операторы цикла с заданным числом повторений

Эти операторы цикла задают заранее известное количество повторений тела. В составе их секций управления имеется так называемая *переменная цикла*. Переменная цикла принимает дискретное количество значений и используется для подсчета количества повторений тела, поэтому иногда ее называют счетчиком. Секция управления ограничивает количество значений переменной цикла, определяя ее начальное, конечное значение, а также величину шага между соседними значениями. Начальное и конечное значения переменной, а также величину шага называют *параметрами цикла*.

Наиболее простой вариант подобного оператора цикла предлагается в языке Pascal. Он записывается в следующем виде:

```
for переменная := начальное_значение to конечное_значение do оператор_тела
```

Секцией управления здесь является конструкция:

```
for переменная := начальное_значение to конечное_значение
```

Опишем семантику данного `for`-оператора. Сначала переменная (счетчик) принимает начальное значение и выполняется тело, затем переменная увеличивается на единицу и тело снова выполняется и так далее, пока значение переменной не превысит конечного значения. В момент превышения цикл прекращается. Заметим, что оператору тела (составному или простому) запрещено изменять значение переменной цикла. Считается, что при выходе из цикла значение переменной цикла не определено.

Для этого оператора обеспечивается и другой вариант: уменьшение переменной цикла на единицу перед каждом повтором. Достаточно лишь применить секцию управления с иным ключевым словом:

```
for переменная := начальное_значение downto конечное_значение
```

При этом условие прекращения цикла приобретает вид:

```
переменная < конечное_значение
```

Отметим также два обстоятельства. Во-первых, условие, управляющее работой for-оператора, проверяется перед выполнением тела. Если условие не выполняется в самом начале работы оператора, тело не будет выполнено ни разу. Во-вторых, величина шага не меняется (является постоянной).

Похожие операторы встречаются и в других языках программирования. Специфику их реализации обычно определяют следующие решения:

- Переменная цикла задается внутри оператора цикла или вне его.
- Параметры цикла вычисляются однократно или многократно.

Однократное вычисление параметров цикла гарантирует простоту реализации оператора, в то время как многократное вычисление существенно увеличивает гибкость оператора.

Оператор for языка Ada создавался на основе своего аналога в языке Pascal. Он тоже является циклом с предварительной проверкой условия. Используются две формы оператора.

Первая форма оператора for имеет вид:

```
for  параметр цикла in  дискретный диапазон  loop
    операторы тела цикла
end loop;
```

Интересно заметить, что здесь параметр цикла — это переменная, которая заранее не описывается (в программе). Данная переменная определена только внутри оператора цикла. Параметру цикла последовательно присваиваются значения из дискретного диапазона. Дискретный диапазон всегда записывается в порядке возрастания:

```
min .. max;
```

Операторы тела повторяются для каждого значения параметра цикла (от минимального до максимального).

Проиллюстрируем работу первой формы оператора цикла примером:

```
for Count in 1 .. 10 loop
    Put ( Count );
end loop;
```

При выполнении цикла на экран выводится:

```
1  2  3  4  5  6  7  8  9  10
```

Вторая форма оператора for выглядит так:

```
for  параметр цикла  in reverse  дискретный диапазон  loop
     операторы тела цикла
end loop;
```

Отличие этой формы состоит в том, что значения параметру присваиваются в порядке убывания (от максимального к минимальному). Диапазон же задается по-прежнему, в порядке возрастания.

В качестве примера запишем фрагмент:

```
for Count in reverse 1 .. 10 loop
    Put ( Count );
end loop;
```

При выполнении этого фрагмента на экран выводится:

```
10  9  8  7  6  5  4  3  2  1
```

Дискретный диапазон в операторе `for` языка Ada вовсе не обязан быть числовым. В общем случае можно говорить, что данный оператор реализует *цикл с повторением по данным*, поскольку именно количество элементов данных определяет количество итераций тела.

Циклы с повторением по данным позволяют перебирать все элементы многоэлементной структуры, обращаясь к каждому из них по очереди; они очень востребованы в объектно-ориентированном программировании, где используются для создания итераторов. Например, в языке C# существует оператор `foreach`:

```
foreach (int x in intArray) {операторы_тела};
```

При каждом повторении цикла целая переменная `x` будет принимать значение, равное следующему элементу массива `intArray`. Размер массива определяет количество повторений тела цикла.

ПРИМЕЧАНИЕ

Как минимум, итератор включает операции для проверки наличия элемента в структуре, доступа к текущему элементу и перемещения к следующему элементу. Цикл for-each достаточно часто используется итератором как базовый механизм, позволяющий обращаться ко всем элементам структуры с небольшими накладными расходами.

Оператор for языка Python

Обобщенная форма оператора `for` в языке Python имеет вид:

```
for переменная_цикла in структура_данных :
    - тело цикла
[else:
    - ветвь else]
```

Переменной цикла последовательно присваиваются значения из структуры данных, которая обычно задается в виде диапазона значений. При каждом значении переменной повторяется тело цикла. Если в операторе присутствует ветвь `else`, то она выполняется при нормальном завершении цикла.

При выполнении цикла

```
for loopVar in [17, 21, 47]:
    print loopVar
```

на экран выводится:

```
17
21
47
```

В большинстве простых циклов по счетчику в этом языке задействуется функция `range`. У функции `range` может быть один, два или три параметра. Проиллюстрируем примерами действия функции `range`:

- `range(4)` возвращает `[0, 1, 2, 3]`;

- `range(2, 9)` возвращает `[2, 3, 4, 5, 6, 7, 8]`;
- `range(0, 10, 2)` возвращает `[0, 2, 4, 6, 8]`.

Обратим внимание, что функция никогда не возвращает последнее значение из заданного диапазона параметров.

Циклы с заданным числом повторений в функциональных языках

В чисто функциональных языках отсутствуют переменные, поэтому вместо итерации здесь используют рекурсию, а именно: рекурсивные функции. Циклы с заданным числом повторений моделируются в функциональных языках следующим образом: счетчик может быть параметром для функции, многократно выполняющей тело цикла. Это тело определяется во вспомогательной функции, которая тоже посылается в функцию цикла как параметр. Таким образом, функция цикла принимает в качестве параметров тело вспомогательной функции и количество повторений.

В языке F# обобщенная форма функции для моделирования циклов по счетчику может называться `loopFor` и имеет следующий вид:

```
let rec loopFor bodyLoop reps =
    if reps <= 0 then
        ( )
    else
        bodyLoop()
        loopFor bodyLoop, (reps - 1);;
```

В этой функции параметр `bodyLoop` является функцией с телом цикла, а параметр `reps` — это количество повторений. Зарезервированное слово `rec`, предваряющее имя функции, определяет рекурсивность функции. Пустые круглые скобки обозначают отсутствие действия, их появление объясняется тем, что пустой оператор запрещен, а каждая конструкция `if` должна иметь ветвь `else`.

Операторы цикла без заданного числа повторений

Для этой категории операторов цикла количество повторений тела заранее не известно. Решение о прекращении цикла принимается по результатам анализа текущей ситуации. В этом случае говорят о логически управляемых циклах. Подобные циклы соответствуют итерациям более общего вида: любой цикл с управлением по счетчику можно построить с помощью логически управляемого цикла, но не наоборот.

Бесконечные циклы и механизмы управления ими

Иногда условия выхода из цикла неизвестны и не могут быть выражены в секциях управления операторов цикла. В таких ситуациях языки предлагают конструкции бесконечных циклов. Например, в языке Ada бесконечный цикл имеет вид:

```
loop
    операторы тела
end loop;
```

В языке Pascal бесконечный цикл организуется с помощью оператора `while` с условием, которое всегда выполняется:

```
while true do begin операторы тела end
```

Аналогичную роль играет следующая конструкция языка C:

```
for(; ;){ операторы тела }
```

В подобных случаях программисту приходится встраивать в тело свой собственный механизм управления циклом.

Для языка Ada встраивание механизма управления может принять следующую форму:

```
loop
    последовательность операторов 1
    exit when условие выхода
    последовательность операторов 2
end loop;
```

Поясним порядок работы этого цикла:

1. Выполняется последовательность операторов 1.
2. Вычисляется значение условия выхода. Если значение равно `True`, происходит выход из цикла.
3. Выполняется последовательность операторов 2. Осуществляется переход к пункту 1.

ПРИМЕЧАНИЕ

1. Операторы тела повторяются пока условие равно False.
2. В теле должен быть оператор, влияющий на значение условия. Иначе цикл будет выполняться бесконечно.
3. В теле цикла возможно использование безусловного оператора выхода **exit** или условного оператора выхода **exit when** `условие`.

В качестве примера рассмотрим следующий фрагмент:

```
Count := 1;
loop
    Put ( Count );
    exit when Count = 10;
    Count := Count + 1;
end loop;
```

В ходе исполнения фрагмента на экран выводится:

```
1  2  3  4  5  6  7  8  9  10
```

Аналогичные вычисления можно представить и в модифицированном виде:

```
Count := 1;
loop
    Put ( Count );
```

```
    if Count = 10 then
        exit;
    end if;
    Count := Count + 1;
end loop;
```

Здесь мы применили безусловный оператор выхода из бесконечного цикла.

Языки C, C++, C# и Java обеспечивают следующие средства управления циклами:

- Безусловный оператор `break` для выхода из цикла.
- Безусловный оператор `continue` для досрочного прекращения текущей итерации с передачей управления на следующую итерацию цикла.

Следующий фрагмент показывает пример использования операторов `break` и `continue` внутри циклов:

```
секция_управления_циклом {
    c = getchar ( );
    if (условие_прекращения_итерации)
        continue; /* начать новую итерацию */
    if (условие_выхода)
        break; /* выйти из цикла */
    ++ linecount;
    }
```

Секция управления циклом здесь не конкретизирована, записано лишь тело цикла. Первый оператор этого цикла заносит в переменную с именем `c` некоторый символ. Далее анализируется `условие_прекращения_итерации`, и если оно выполняется, оператор `continue` прекращает текущее выполнение тела, передавая управление на следующую итерацию цикла. В противном случае по результатам анализа `условия_выхода` оператор `break` может передать управление за пределы цикла.

Заметим, что в языке Java операторы `break` и `continue` могут включать в себя адрес перехода, помеченный меткой. Метка — это идентификатор с последующим двоеточием:

```
метка:
```

Метка может указывать на любой внешний составной оператор. В этом случае `break` и `continue` становятся подобны оператору `goto`, со всеми вытекающими из этого факта последствиями. Такой механизм может инициировать программирование в стиле «спагетти», нарушающем постулаты структурированного программирования.

Зачем добавлена такая возможность? Рассмотрим пример:

```
loop:
внешний_цикл {
…
    внутренний_цикл {
    …
    if (c1) continue loop;
    …
    if (c2) break loop;
    }
}
```

Прежде всего проясним вопрос с метками. Фактически метку в языке Java следует указывать лишь перед оператором цикла. Такое указание свидетельствует о вложении в цикл другого цикла или оператора выбора. Обычные версии `break` и `continue` прерывают только текущий цикл, в то время как их версии с метками способны досрочно завершать циклы и передавать выполнение в точку, адресуемую меткой.

В нашем примере оператор `continue loop` влечет выход из внутреннего и внешнего циклов и возврат к метке `loop`. Далее выполнение цикла продолжается, но с внешнего цикла. В свою очередь, оператор `break loop` также вызывает переход к метке `loop`, но на этот раз повторный вход в цикл не происходит. Это действие останавливает выполнение обоих операторов цикла.

Конечно, такое решение повышает гибкость программирования, но с точки зрения структурирования программы оно «на грани фола», поскольку способно нарушить соглашение о единственной точке входа и единственной точке выхода структуры управления.

Циклы с предусловием

В этих операторах цикла секция управления размещается перед телом. Наиболее популярной конструкцией цикла с предусловием считается оператор `while` (язык Pascal):

```
while условие продолжения do тело
```

Тело повторяется, пока условие имеет истинное значение.

В языке Ada оператор цикла `while` имеет следующий вид:

```
while условие продолжения loop
    последовательность операторов
end loop;
```

Порядок выполнения оператора:

1. Вычисляется значение условия. Если значение равно `True`, выполняется переход к пункту 2. В противном случае (при значении `False`) происходит выход из цикла.
2. Выполняются операторы тела цикла. Осуществляется переход к пункту 1.

Таким образом, это цикл с условием продолжения, предваряющим исполнение тела.

Отметим характерные особенности оператора `while`:

1. Операторы тела могут выполняться нуль и более раз.
2. Операторы тела повторяются пока условие равно `True`.
3. В теле должен быть оператор, влияющий на значение условия (для исключения бесконечного повторения).

Рассмотрим пример:

```
Count := 1;
while Count <= 10 loop
    Put ( Count );
    Count := Count + 1;
end loop;
```

При выполнении цикла на экран выводится:

```
1  2  3  4  5  6  7  8  9  10
```

Синтаксис оператора `while` в языках C, C++, C# и Java не предусматривает ключевого слова `do`, но обязывает ставить скобки вокруг условия продолжения:

```
while (условие продолжения) тело
```

Дело в том, что в C-подобных языках отсутствуют служебные слова `begin`, `then`, `end`.

Например, можно записать:

```
while (i != c) {
    i = i – 1;
    y += a;
}
```

Заметим, что условие продолжения оператора `while` в языках C, C++, C# является целочисленным (единица моделирует истину), а в языке Java — должно быть булевым.

Еще один пример. Выражение `c = getchar( )` назначает переменной с именем `c` набираемый на клавиатуре символ, который читается стандартной функцией `getchar( )`. Значение, назначенное переменной `c`, становится также значением выражения *c* = `getchar( )`, которое может появиться как подвыражение внутри большого выражения

```
(c = getchar( )) != EOF
```

Константа EOF обозначает конец ввода. Это выражение читает значение, используя `getchar( )`, сохраняет значение путем присвоения его переменной `c`, а затем проверяет — не равно ли значение константе EOF. Иногда такие выражения используются для проверок в `while`-циклах:

```
while ((c = getchar( )) != EOF) putchar (c);
```

Циклы с предусловием могут моделироваться в чисто функциональной форме с помощью рекурсивной функции. Специфика моделирования схожа с той, которая рассматривалась в подразделе «Циклы с заданным числом повторений в функциональных языках». В обоих случаях тело цикла записывается в виде функции. Представим обобщенную форму для моделирования цикла с предусловием, написанную на языке F#:

```
else

let rec loopWhile condition bodyLoop =
    if  condition() then
        bodyLoop()
        loopWhile condition bodyLoop
    else
        ( );;
```

Циклы с постусловием

В этой разновидности операторов секция управления размещается после тела. Можно вообразить ситуацию, когда проверку условия продолжения следует проводить после выполнения тела цикла. В таких ситуациях языки C, C++, C# и Java рекомендуют оператор `do-while`:

```
do
    тело цикла
while (условие продолжения)
```

Тело цикла с последующей проверкой условия повторяется до тех пор, пока условие не станет ложным (нулем). Единственное различие между операторами `do-while` и `while` состоит в том, что оператор do-while всегда заставляет тело цикла выполняться хотя бы один раз. В обоих случаях оператор тела является составным.

При применении оператора `do-while` предыдущий пример надо переписать в следующей форме:

```
do {
    i = i - 1;
    y += a;
} while (i != c)
```

Если условие выполняется, управление передается на другую итерацию цикла.

Оператор цикла с постусловием `repeat-until` языка Pascal имеет следующий синтаксис:

```
repeat
    тело цикла
until условие окончания
```

Тело цикла повторяется до тех пор, пока условие окончания не установится в значение `True`. Таким образом, логика управления оператора `repeat-until` противоположна логике управления оператора `do-while` языков C, C++, C# и Java.

Проиллюстрируем сказанное, переписав предыдущий пример в терминах оператора `repeat-until`:

```
repeat
    i := i - 1;
    y := y + a
until i = c
```

Если условие выполняется, происходит выход из цикла.

ПРИМЕЧАНИЕ

1. Смысл оператора repeat S until E; можно передать строкой
   `S; while not E do S`
2. Смысл оператора while E do S; можно передать строкой
   `if E then repeat S until not E`

В общем случае цикл с постусловием считают менее полезным и даже опасным, поскольку программисты часто забывают о том, что его тело всегда должно выполняться хотя бы один раз.

Универсальность оператора for в языках C, C++,C# и Java

В общем случае оператор цикла `for` для C-подобных языков задает не определенное заранее количество повторений. Этим он сильно отличается от своих «собратьев» в языках Pascal, Ada.

Оператор `for` языка C имеет следующий вид:

```
for (выражение_1; выражение_2; выражение_3)
    тело цикла
```

Как видим, секция управления содержит три выражения:

- `выражение_1` — вычисляется перед входом в цикл;
- `выражение_2` — определяет условие продолжения цикла;
- `выражение_3` — вычисляется перед каждой следующей итерацией цикла.

Внутри всех выражений могут содержаться присваивания. Первое выражение может использоваться для инициализации цикла и вычисляется только один раз в начале выполнения оператора `for`. Второе выражение предназначено для управления повторениями, оно вычисляется перед каждым выполнением тела цикла. Как это принято в языке C, нулевое значение означает *ложь*, а все ненулевые значения означают *истину*. Если значение второго выражения равно нулю, то происходит выход из цикла; в противном случае тело цикла повторяется. Третье выражение применяется для подготовки следующей итерации и вычисляется после каждого выполнения тела цикла. Например, оно может обеспечивать увеличение счетчика цикла.

Тело цикла `for` может включать в себя отдельный оператор, составной оператор или быть пустым.

Оператор `for` — универсальный и мощный инструмент. Например, с его помощью цикл

```
while (x != A[i])   i = i - 1;
```

может быть переписан как

```
for (; x != A[i]; i = i - 1);
```

Здесь нет инициализации, а присваивание `i = i - 1` выполняется перед следующей итерацией. Тело цикла пусто. Оператор просто не нуждается в теле цикла, а потому и не имеет его. Все требуемые действия являются частью секции управления, а не его тела.

В секции управления для `for`-оператора все выражения необязательны. Отсутствие второго выражения рассматривается как его истинное значение. Поэтому запись `for(; ;)` задает бесконечный цикл. Заметим, что по поводу отсутствия первого и третьего выражений никакие предположения не делаются.

Сам по себе оператор `for` не нуждается в счетчике. Основные идеи его создания состояли в следующем:

- не существует никаких явных счетчиков или параметров цикла;
- все переменные, указанные в секции управления, могут изменяться в теле цикла.

Реализация этих идей позволила оператору `for` легко моделировать как подсчет, так и любую другую логику управления повторением (хотя и за счет некоторого понижения надежности вычислений).

Приведем типичный фрагмент кода, в котором используется цикл со счетчиком на основе оператора `for`:

```
sum = 0;
for (i = 0; i <= 100; i++)
    sum = sum + vector[i];
```

Еще одним основанием высокой гибкости оператора `for` стала возможность использования составных выражений. Более конкретно: составные выражения разрешают применять в качестве первого и третьего выражений для секции управления. Составное выражение формируется последовательностью простых выражений, разделяемых запятыми. Значения этих выражений вычисляются в порядке их следования, то есть слева направо. Рассмотрим пример реверсирования элементов в строке `str`, имеющей длину `m`:

```
int buf, i, j;
for (i = 0, j = m - 1; i < j; i++, j--) {
    buf = str[i];
    str[i] = str[j];
    str[j] = buf;
}
```

В ходе реверсирования первый элемент замещается последним, второй элемент — предпоследним и т. д. Инициализация цикла и подготовка к новой итерации здесь задаются составными выражениями.

Оператор `for` языка C++ отличается от своего прообраза в языке C двумя дополнениями. Во-первых, для управления циклом он может использовать как арифметическое, так и булево выражение. Во-вторых, первое выражение может содержать определения переменных. Например, счетчик можно объявить как целую переменную:

```
for (int count = 0; count < max; count++) {…}
```

Область действия параметра `count` распространяется на тело цикла.

Операторы `for` для языков Java и C# аналогичны оператору `for` языка C++, однако в них для управления повтором разрешены только булевы выражения.

Охраняемые структуры управления Дейкстры

В основе структур управления ветвлением и циклами, созданных Э. Дейкстрой, лежало стремление гарантировать правильность потока управления [11, 61].

Он предложил следующую структуру охраняемого ветвления:

```
if предохранитель -> последовательность_операторов
[] предохранитель -> последовательность_операторов
[] …
[] предохранитель -> последовательность_операторов
fi
```

Замыкающая скобка `fi` представляет собой открывающую скобку `if`, записанную наоборот. Маленькие блоки `[]`, названные барьерами, используются для отделения охраняемых операторов. Каждая строка в структуре называется охраняемой командой.

Эта структура ветвления имеет следующую семантику. Все предохранители (булевы выражения) вычисляются каждый раз при достижении этой структуры в программе. Если истинными являются несколько предохранителей, то для выполнения выбирается (случайным образом) одна из последовательностей операторов. Если

ни один предохранитель не имеет истинного значения, вычисления прекращаются. Такой подход вынуждает программиста рассматривать все возможности ветвления. Рассмотрим следующий пример:

```
if i = 0 -> rez := rez + x
[] i > j -> rez := rez + y
[] j > i -> rez := rez + x
fi
```

Если `i = 0` и `j > i`, то эта структура случайным образом выбирает первый или третий операторы присваивания. Если `i = j` и `i < > 0`, то возникает ошибка вычислений.

С помощью этой структуры программист указывает: порядок вычислений (в некоторых случаях) значения не имеет. Например, поиск наибольшего из двух чисел записывается в виде:

```
if a >= b -> max = a
[] b >= a -> max = b
fi
```

Здесь описывается желаемый результат, но не навязывается алгоритм решения. В частности, если `a = b`, то неважно, какое именно значение заносится в переменную `max`. Очевидно, что семантика этой структуры недетерминирована.

Запрограммируем эту задачу с помощью обычного условного оператора:

```
if (a >= b)
    max = a;
else
    max = b;
```

Впрочем, возможно и другое решение:

```
if (a > b)
    max = a;
else
    max = b;
```

С точки зрения практики разницы между этими решениями нет. В первом случае переменной `max` назначается значение `a` (при `a = b`), во втором случае — значение `b`. По сути проблема выбора одного из вариантов лишь усложняет формальный анализ и доказательство правильности программного кода. В этом и состоит одна из причин разработки Дейкстрой охраняемых команд.

Охраняемый цикл Дейкстры имеет следующую структуру:

```
do предохранитель -> последовательность_операторов
[] предохранитель -> последовательность_операторов
[] …
od
```

Семантика этой структуры заключается в том, что все предохранители вычисляются при каждом повторении цикла. Если истинными оказываются несколько предохранителей, то связанные с ними операторы сначала выбираются случайным образом, а затем и выполняются. После выполнения снова вычисляются все предохранители. Когда все предохранители одновременно станут ложными, цикл завершается.

Рассмотрим следующую задачу: упорядочить значения четырех целочисленных переменных `a`, `b`, `c`, `d` так, чтобы выполнялось условие $a \le b \le c \le d$. Прямолинейный подход (без охраняемых команд) состоит в занесении значений в массив, сортировке

массива и записи отсортированных значений в переменные. Сделать это нетрудно, но потребуется достаточно длинная программа, реализующая, в том числе, какой-то метод сортировки.

С другой стороны, охраняемые команды обеспечивают более короткое и элегантное решение:

```
do a > b -> temp = a; a = b; b = temp;
[] b > c -> temp = b; b = c; c = temp;
[] c > d -> temp = c; c = d; d = temp;
od
```

Охраняемые команды Дейкстры являются наглядной иллюстрацией влияния синтаксиса и семантики операторов на проверку правильности программ (и обратного влияния). Многочисленные исследования доказали, что проверка правильности программ очень затруднительна, если используются операторы безусловного перехода. Проверка существенно упрощается, если используются только классические структурированные средства управления либо только охраняемые команды. Однако, к сожалению, сложность реализации охраняемых команд очень высока.

Инварианты

Инварианты являются ключевым понятием при разработке императивных программ.

Инвариантом в некоторой точке программы является утверждение, которое сохраняет свой смысл при достижении этой точки во время выполнения программы, то есть при достижении управлением этой точки. По сути, утверждение — это условие, которое в зависимости от состояния вычислений принимает значение *истина* или *ложь*. Пример — условие `x > y`, которое относится к значениям величин `x` и `y`.

Главная трудность в написании корректной программы состоит в том, что правильность — это свойство не ее статического текста, а свойство динамических вычислений, проводимых программой в процессе работы.

Инварианты помогают судить о вычислениях программы, играют роль моста между статическим программным текстом и динамическим процессом вычислений.

Когда мы говорим об инвариантах, на самом деле мы работаем с утверждениями, поскольку имеем дело со свойствами, которые сохраняются при достижении управлением конкретной программной точки.

Плохой способ — написать программу, а затем модифицировать инварианты. Лучший способ — начать с инвариантов и использовать их для разработки программы.

Например, разработка программы, устраняющей смежные дублирования, может стартовать с инварианта:

```
{здесь: x — первый элемент данных из анализируемой серии}
```

ПРИМЕЧАНИЕ

Как это принято в аксиоматическом подходе, инварианты будем помещать в фигурные скобки.

Продолжим создание программы. Выведем элемент на экран, а затем многократно читаем элементы, пропуская дубликаты x. Читаемые элементы заносим в переменную next до тех пор, пока значение next не станет отлично от значения x:

```
writeln (x);
repeat read (next) until next <> x;
{здесь: несколько раз прочитали элементы с одним и тем же значением}
```

Элемент с новым значением заносим в x:

```
x := next;
{здесь: организуем новую серию действий по устранению дубликатов}
```

Конечная версия программы примет вид:

```
read (x);
while x <> маркер конца списка do begin
    {здесь: x — первый элемент данных из анализируемой серии}
    writeln (x);
    repeat read (next) until next <> x;
    {здесь: несколько раз прочитали элементы с одним и тем же значением}
    x := next;
    {здесь: организуем новую серию действий по устранению дубликатов}
end
```

Программирование с инвариантами

При использовании конструкций типа «один вход / один выход» поток управления в программе очевиден из ее текста.

Как это может помочь разработке программы?

Если оператор S имеет один вход и один выход, то его поведение может быть задано двумя условиями:

1) условием, выполняемым перед входом в оператор (*предусловием*);
2) условием, выполняемым после выхода из оператора (*постусловием*).

Иными словами, поведение оператора определяется *инвариантными утверждениями*.

Например: для цикла WHILE эти утверждения должны формулировать:

- предусловие — условия, требуемые для выполнения цикла;
- инвариант внутри тела цикла — условия, требуемые для нахождения в цикле;
- постусловие — условия, справедливые по выходу из цикла.

Рассмотрим оператор:

```
while x ≥ y do
    x := x - y;
```

Очевидно, что каждый раз перед присваиванием x := x - y должно выполняться x ≥ y.

Запишем это инвариантное утверждение:

```
while x ≥ y do
    {x ≥ y, если мы здесь}
     x := x - y;
```

Предположим, что перед входом в цикл справедлив инвариант (предусловие):

```
{x ≥ 0 and y > 0}
```

Проверка выражения `x ≥ y` между `while` и `do` будет гарантировать, что `x ≥ y` перед присваиванием `x := x – y`.

```
{x ≥ 0 and y > 0}
while x ≥ y do
    {y > 0 and x ≥ y}
     x := x – y;
```

После присваивания измененное значение `x` должно удовлетворять условию `x ≥ 0`.

Доказательство

Положим, что старое значение `x` равно `a`. Из `y > 0` и `a ≥ y` следует, что `a – y ≥ 0`. Поскольку измененное значение `x = a – y` (после присваивания), мы получаем `x ≥ 0`.

Таким образом, записываем:

```
{x ≥ 0 and y > 0}
while x ≥ y do
    {y > 0 and x ≥ y}
     x := x – y;
    {x ≥ 0 and y > 0}
```

Управление опять передается на заголовок. После присваивания инвариант сохраняется для следующей итерации.

Пример. Разработка программы для организации линейного поиска

Об инвариантах думают до написания программы. Тогда инварианты могут быть использованы для руководства программной разработкой. Используем инварианты для разработки программы поиска элементов в массиве (таблице).

Массив *A* может иметь следующую структуру, представленную на рис. 5.2.

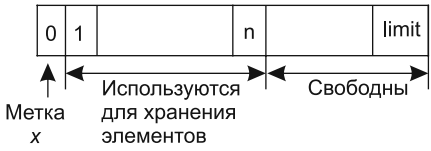


Рис. 5.2. Структура массива для линейного поиска

В массиве (таблице) определены две операции `insert (x)` и `find (x)`. Вначале массив пуст и `n = 0`. Элементы вставляются слева направо, начиная с позиции `1`. То есть вставляемыми элементами являются `A[1], A[2], …A[n]`. Массив обеспечивает следующий инвариант:

```
{Элементы в массиве принадлежат подмассиву A[1..n], при этом 0 ≤ n ≤ limit}
```

Рассмотрим реализацию операции `find (x)`. Операция должна вернуть позицию элемента `x` в массиве. Для упрощения логики поиска условимся:

- Вначале в нулевую позицию массива заносится `x`: `A[0] := x` (как сигнальная метка). Это предполагает, что поиск всегда заканчивается успешно, возвратом

позиции `i`. Если `i = 0`, то элемента нет в массиве. Если `i > 0`, то элемент в массиве есть и `i` — номер позиции элемента.

- Массив просматривается справа налево, двигаясь от последнего вставленного элемента к первому элементу.

Тогда постусловие для поиска записывается в виде:

```
do поиск;
{x = A[i] and x нет в A[i+1 .. n] and 0 ≤ i ≤ n},
```

а начальный текст может иметь следующую форму:

```
инициализация;
{предусловие}
while еще можно искать do
    i:= i – 1;
{постусловие:  x = A[i] and x нет в A[i+1 .. n] and 0 ≤ i ≤ n}
return i;
```

Если показать инициализацию как установку сигнальной метки и как установку начальной позиции поиска, то предусловие примет вид:

```
A[0] := x; i := n;
{x = A[0] and x нет в A[i+1 .. n] and 0 ≤ i ≤ n}
while еще можно искать do
    i := i – 1;
{x = A[i] and x нет в A[i+1 .. n] and 0 ≤ i ≤ n}
return i;
```

Поиск останавливается, если `x` найден или если вышли за пределы массива `[x = A[i]` или `i = 0]`. Исходя из постусловия, оба условия сжимаются до `x = A[i]`. Таким образом, условие нахождения внутри цикла равно `x ≠ A[i]`, а конечный текст имеет вид:

```
A[0] := x; i := n;
{предусловие}
while x ≠ A[i] do
    i := i – 1;
{постусловие}
return i;
```

Контрольные вопросы и упражнения

1. Перепишите следующие фрагменты программ так, чтобы вложенные `if`-операторы *не размещались* внутри `then`-частей внешних `if`-операторов. Вместо этого примените форму `elsif`. Пожалуйста, не стесняйтесь переделывать любые булевы выражения в эквивалентные формы. Используйте нулевые или пустые операторы.

```
а)  if c <= 'I' then
        if c < 'I' then S1
    else S2
б)  if count < max then
        if count > min then S1
        else S2
        end
    end
```

```
в)  if n >= 60 then
        if n < 80 then
              if n >= 70 then seventy (n)
              else sixty (n)
              end
        else eighty (n)
        end
    else twenty (n)
    end
```

2. Нарисуйте схемы алгоритмов для следующих программных фрагментов:

```
а)  if E1 then S1
    elsif E2 then S2
    else S3
    end
б)  repeat S until E
в)  loop
        S1;
            if E then exit end;
        S2;
    end
г)  repeat
        S1;
        if E then
            done := true
        else
            S2
    until done
```

3. *Дано*: подмассив `A[i..n]`. *Требуется*: найти `k`-е вхождение элемента `x` (`k > 0`). Использовать инварианты.

   Идея: Элемент `x` должен встречаться в массиве `k` раз. Область поиска последовательно сужается слева–направо.

```
Инвариант цикла:
{должен быть (область) and счетчик < k}
Постусловие:
Начальный текст программы:
Инициализация (обнуление счетчика вхождений,
               установка начальной левой границы области)
Цикл
{должен быть (область) and счетчик < k}
Если область пуста, то выход;
Если элемент массива = x, то увеличить счетчик;
Если счетчик = k, то вернуть индекс элемента и выйти;
(Перейти к следующему элементу). Сужение области поиска.
```

   Продолжить самостоятельно.

4. Какие недостатки имеются у арифметического оператора `IF` языка Fortran?

5. Что необычного есть в операторе выбора языка C? Какие компромиссы были приняты при его проектировании?

6. Какие проблемы возникают при разработке циклов с известным числом повторений?

7. Что представляет собой цикл с предусловием? Что представляет собой цикл с постусловием?
8. В чем заключается разница между оператором `for` языка C и аналогичным оператором языка C++?
9. Какова основная причина изобретения циклов с повторением по данным?
10. Какие преимущества имеет оператор `exit` в языке Ada над оператором `break` языка C?
11. В чем состоит разница между оператором `break` языка C++ и его аналогом в языке Java?
12. Перепишите следующий фрагмент программы на языке C, не используя операторы `goto` и `break`.

```
j = -5;
for (i=0; i < 5; i++) {
    switch ( j + 1 ) {
        case 5:
        case 2; j--; break;
        case 0: j +=1; break;
        default: j = 0;
    }
    if (j > 0) break;
    j = 5 – i
}
```

Глава 6

Средства представления синтаксиса языков программирования

Данная глава посвящена рассмотрению следующих тем. Во-первых, характеризуются особенности определения языка программирования, показатели качества его синтаксиса. Затем подробно обсуждаются синтаксические элементы языка, после чего вводится понятие абстрактного синтаксиса и абстрактных синтаксических деревьев. Далее поясняется термин «контекстно-свободная грамматика» (как средство для точного и формального задания синтаксиса) и описываются самые популярные формы для представления грамматики: обычная и расширенная формы Бэкуса—Наура, а также синтаксические схемы. Исследуются основные концепции, закладываемые в основу создания новой грамматики. Демонстрируется применение двух инструментов для выполнения грамматического разбора: графических деревьев разбора и текстовых выводов. Приводятся наглядные примеры выявления синтаксических проблем: неоднозначности повисшего `else`, стороннего влияния таких элементов, как точки с запятой и пустые операторы.

Особенности определения языка программирования

Определить краткое и понятное описание языка программирования достаточно сложно, но необходимо для обеспечения его практического применения.

Автор языка программирования должен найти способ наглядного представления операторов, выражений и других программных конструкций языка. Популярность языка напрямую связана с ясностью и точностью его описания. Если хотите, нужно, чтобы программисты поймали кураж определения языка, чтобы им было легко и просто его использовать. При описании обычно выделяют задачу формирования синтаксиса и семантики. Синтаксис языка программирования определяет форму, внешний вид конструкций, а семантика придает этим конструкциям смысл. Например, оператор `if` языка C++ имеет следующий синтаксис:

```
if (<выражение>) <оператор>
```

Его семантика заключается в том, что если выражение имеет истинное значение, то будет выполнен указанный оператор.

Для удобства рассуждений семантика и синтаксис часто разделяются, но, по сути, они — две стороны одной медали. В хорошо определенном языке программирования семантика непосредственно следует из синтаксиса; то есть форма конструкции должна способствовать раскрытию ее смысла. Описывать синтаксис намного легче, чем семантику. Причин несколько. Во-первых, форма всегда проще содержания. Во-вторых, существует краткий и общепринятый способ записи синтаксиса, тогда как для описания семантики подобного способа пока еще не придумали.

По форме программы представляют собой совокупности строк, состоящих из символов некоторого алфавита. Правила синтаксиса языка определяют, какие именно строки могут быть названы корректными. Например, в языке C++ оператор `A = B + C` считается правильной последовательностью символов, а выражение `B C +` не соответствует правильной последовательности.

Синтаксис определяет важную информацию, необходимую как для понимания программы, так и для ее трансляции в исполняемый код. Например, синтаксис строки $5 + 6 \times 7$ задает единственную правильную интерпретацию вычислений как $5 + (6 \times 7)$, приводящую к результату 47.

Вместе с тем синтаксической формы строки может оказаться недостаточно для раскрытия смысла вычисления. Например, в строке `Y = 4.47+3.14` синтаксис не подскажет, какой следует считать переменную `Y` и операцию сложения «`+`». Они могут быть как целыми, так и вещественными. При целой переменной `Y` и целочисленном сложении результат будет `Y = 7`. При целой переменной `Y` и вещественном сложении получится итог `Y = 8`. Наконец, для вещественной переменной и вещественного сложения результат примет вид `Y = 7.61`. Для выбора варианта вычисления нам дополнительно потребуется синтаксическое определение имени `Y` и семантическая интерпретация смысла значка «`+`».

В качестве назидательного примера напомним широко известную проблему с синтаксисом языка Fortran. Во всех языках программирования группы символов (слова) в программе отделяются пробелами, однако в языке Fortran пробельные символы игнорируются. Рассмотрим оператор цикла, определяющий «цикл до метки 10 при изменении индекса i от 1 до 100»:

```
do 10 i = 1,100
```

Если запятая случайно заменена точкой, то этот оператор истолковывается как оператор присваивания значения 1.100 той переменной, имя которой образуется соединением всех символов перед значком «=»:

```
do10i = 1.100
```

Считают, что эта ошибка заставила ракету с космическим аппаратом «Маринер-2» взорваться на старте!

Для простоты синтаксические определения языков программирования обычно не содержат развернутые описания единиц самого нижнего уровня. Такие элементарные единицы называют лексемами. Описание лексем обычно задается лексическим синтаксисом языка. В число лексем языка входят, например, его идентификаторы, литеральные константы, символы операций и специальные слова. Удобно считать, что программа заключает в себе строки, состоящие не из символов, а из лексем.

Приводимые в этой главе средства представления синтаксиса языков очень просты и содержат описания лексем.

Хотя правил синтаксиса явно недостаточно для полного понимания языка программирования, все же синтаксис считается очень важным элементом языка. В целом, определение синтаксиса является решенной задачей. Трансляторы автоматически создают полное синтаксическое описание текста исходной программы. Использование же семантики для генерации исполняемого кода все еще требует большого мастерства в сочетании с применением отдельных формальных методов.

Качество синтаксиса языка

В общем случае большинство программных конструкций может быть записано с помощью нескольких форм. Возникает задача выбора лучшей синтаксической формы. Выбор осуществляют из множества соображений, среди которых выделяют легкость чтения, написания и трансляции программы, а также однозначность [3, 31, 32, 84, 86, 88].

Легкость чтения

Текст легкой для чтения программы делает очевидной структуру и содержание ее алгоритма, а также используемых данных. В идеальном варианте такая программа может стать *самодокументируемой*, то есть не требующей дополнительных комментариев. Легкость чтения достигается за счет естественных форматов операторов, их структурированности, разумного применения ключевых слов, встраивания комментариев, неограниченности длины идентификаторов, мнемонических символов операций и явного объявления всех данных. Конечно, небрежное программирование может свести к нулю всю эту легкость, но, по крайней мере, правила синтаксиса гарантируют удобочитаемость программных текстов. Наибольшую легкость чтения демонстрирует язык Ada, предложения которого, по мнению авторов, близки к текстам на английском языке.

Читабельность программы улучшает такой синтаксис языка, в котором формы конструкций ясно отражают их семантические особенности. Подразумевается, что конструкции с похожей семантикой должны выглядеть подобно, а конструкции, которые выполняют различные действия, должны иметь различные формы. В целом, разнообразие синтаксических конструкций должно соответствовать разнообразию задаваемых действий и объявлений.

Программы, написанные на языках со скудным набором синтаксических конструкций, обычно трудны для чтения. Если в языке одна синтаксическая единица обслуживает много семантических действий (языки APL, SNOBOL4), разобраться в написанной программе бывает очень не просто. Фактически приходится заниматься расшифровкой программного текста. Мало того, искажение одного символа в операторе может полностью изменить его смысл, хотя сам оператор остается синтаксически верным. В функциональных языках к таким последствиям приводит «жонглирование» открывающими и закрывающими скобками.

Легкость написания

Обычно легкость написания программ вступает в противоречие с легкостью их чтения. Дело в том, что писать программу удобно с помощью простых и однородных синтаксических конструкций, а для удобочитаемости требуется их разнообразие. Например, на языке C программы можно записывать очень лаконично, но читать их будет весьма сложно. Такое положение дел приходится исправлять за счет специального стиля оформления текста программы.

Неявные соглашения синтаксиса, исключающие необходимость определения операций и типов переменных, упрощают и укорачивают текст программы, но усложняют его чтение. Компромисс возможен при применении естественных форматов операторов, их структурировании, использовании понятных мнемонических обозначений для операций и неограниченных по длине идентификаторов. Эти средства упрощают как внешний вид, так и смысл программ.

Синтаксис считают *избыточным*, если одни и те же сведения можно записать несколькими способами. Разумная степень избыточности улучшает легкость чтения программы, упрощает обнаружение ошибок во время компиляции. С другой стороны, избыточность синтаксиса может приводить к удлинению текста программ и увеличению времени на их написание. Для уменьшения избыточности синтаксиса обычно применяют следующее правило.

Если какое-либо объявление можно вывести из контекста программы, то от явной записи этого объявления можно отказаться.

Например, в функциональном языке ML вместо явного задания типа для каждого параметра функции эти типы выводятся по индукции. Правда, при появлении ошибки в написании такой функции транслятор не сможет ее обнаружить. Следствием такой маскировки ошибок являются трудности в использовании языков, полностью «избавленных» от избыточности.

Легкость трансляции

Синтаксис языка призван также обеспечивать легкость трансляции текста программы в исполняемый код. Этот показатель качества конфликтует с легкостью чтения и написания программ, которые важны прежде всего для программистов и пользователей. Легкость трансляции важна для транслятора, обрабатывающего написанную программу. В первую очередь трудоемкость трансляции определяется регулярностью структуры программы. Например, программы языка LISP трудны для написания и чтения, но очень регулярны, а потому просты для трансляции. Благодаря регулярности синтаксиса структура любой программы на LISP задается несколькими простыми правилами. Большое количество синтаксических конструкций прямо влияет на сложность трансляции. В частности, из-за большого количества синтаксических правил достаточно сложно транслировать программы на языке Ada, хотя для этого языка характерны как легкость написания, так и легкость чтения.

Отсутствие неоднозначности

Непреложное правило любого языка программирования состоит в том, что определение синтаксиса *должно* обеспечивать однозначность любой допустимой конструкции. Неоднозначная конструкция допускает несколько толкований. Обычно неоднозначность возникает не в отдельно взятом элементе программы, а при взаимодействии различных конструкций. Например, в главе 5 мы обсуждали проблему с *повисшим else*, которая появляется при вложении условных операторов друг в друга:

```
if булево_выражение1 then if булево_выражение2 then оператор1 else оператор2
```

Эта структура является неоднозначной, так как неясно, к какому из `if` следует отнести `else`. В языке Fortran строка `M(I,J)` может означать как элемент двумерного массива `M`, так и вызов подпрограммы `M` с двумя параметрами `I` и `J`. В этом языке в обоих случаях применяется одна и та же синтаксическая конструкция. Примеры синтаксической неоднозначности имеются в любом языке программирования.

Все синтаксические неоднозначности обязательно устраняются. Пути решения, правда, выбираются разные:

- в языке Ada каждый условный оператор `if` должен завершаться закрывающей скобкой `end if`;
- в языках C и Pascal `else` сопоставляется ближайшему к нему `if`;
- в языке Fortran конструкция `M(I,J)` рассматривается как вызов подпрограммы, если не был объявлен массив `M` (ищется предварительное объявление массива `M`);
- в языке Pascal для последнего случая вводится синтаксическое отличие (индексы элемента массива заключаются в квадратные скобки `M[I,J]`, а список параметров при обращении к подпрограмме заносится в круглые скобки `M(I,J)`).

Синтаксические элементы языка

Выразительность синтаксиса языка определяется набором используемых синтаксических элементов. Обсудим наиболее характерные элементы для описания синтаксиса.

Набор символов

Набор используемых символов оказывает решающее воздействие на все остальные строительные элементы синтаксиса. До недавнего времени наиболее распространенным считался набор под именем «американский стандартный код для обмена информацией» ASCII (American Standard Code for Information Interchange). Стандартный код ASCII — 7-разрядный, восьмая позиция отводится для записи бита четности. Это обеспечивает представление 128 символов, включая все латинские буквы, цифры, знаки основных математических операций и знаки пунктуации. Позже появилась европейская модификация ASCII, называемая Latin 1 (стандарт ISO 8859-1). В ней «полезно» используются все 8 разрядов. Дополнительные комбинации (коды 128–255) в новом варианте отводятся для представления специфических букв алфавитов западноевропейских языков, символов псевдографики,

некоторых букв греческого алфавита, а также ряда математических и финансовых символов. Именно эта кодовая таблица считается мировым стандартом де-факто, который применяется с различными модификациями во всех странах. В зависимости от использования кодов 128–255 различают несколько вариантов стандарта ISO 8859 (табл. 6.1).

Таблица 6.1. Варианты стандарта ISO 8859

Стандарт	Характеристика
ISO 8859-1	Западноевропейские языки
ISO 8859-2	Языки стран центральной и восточной Европы
ISO 8859-3	Языки стран южной Европы, мальтийский и эсперанто
ISO 8859-4	Языки стран северной Европы
ISO 8859-5	Языки славянских стран с символами кириллицы
ISO 8859-6	Арабский язык
ISO 8859-7	Современный греческий язык
ISO 8859-8	Языки иврит и идиш
ISO 8859-9	Турецкий язык
ISO 8859-10	Языки стран северной Европы (лапландский, исландский)
ISO 8859-11	Тайский язык
ISO 8859-13	Языки балтийских стран
ISO 8859-14	Кельтский язык
ISO 8859-15	Комбинированная таблица для европейских языков
ISO 8859-16	Содержит специфические символы ряда языков: албанского, хорватского, английского, финского, французского, немецкого, венгерского, ирландского, итальянского, польского, румынского и словенского

Хотя код ASCII достаточно удобен, он все же слишком тесен и не вмещает множества необходимых символов. По этой причине в 1993 году консорциумом компаний Apple Computer, Microsoft, Hewlett–Packard, DEC и IBM был разработан 16-битовый стандарт ISO 10646, определяющий универсальный набор символов (UCS, Universal Character Set). Новый код, известный под названием Unicode, позволяет задать до 65 536 символов, то есть дает возможность одновременно представить символы всех основных «живых» и «мертвых» языков. Для букв русского языка выделены коды 1040–1093.

Все символы в Unicode логически разделяют на 17 плоскостей по 65 536 (2^{16}) кодов в каждой:

- Плоскость 0 (0000–FFFF): BMP, Basic Multilingual Plane — основная многоязычная плоскость. Эта плоскость охватывает коды большинства основных символов используемых в настоящее время языков. Каждый символ представляется 16-разрядным кодом.
- Плоскость 1 (10000–1FFFF): SMP, Supplementary Multilingual Plane — дополнительная многоязычная плоскость.
- Плоскость 2 (20000–2FFFF): SIP, Supplementary Ideographic Plane — дополнительная идеографическая плоскость.
- Плоскости с 3 по 13 (30000–DFFF) пока не используются.

- Плоскость 14 (E0000–EFFFF): SSP, Supplementary Special-purpose Plane — специализированная дополнительная плоскость.
- Плоскость 15 (F0000–FFFFF): PUA, Private Use Area — область для частного использования.
- Плоскость 16 (100000–10FFFF): PUA, Private Use Area — область для частного использования.

В настоящее время используется порядка 10% потенциального пространства кодов.

В «естественном» варианте кодировки Unicode, известном как UCS-2, каждый символ описывается двумя последовательными байтами m и n, так что номеру символа соответствует численное значение $256 \times m + n$. Таким образом, кодовый номер представлен 16-разрядным двоичным числом.

Наряду с UCS-2 в рамках Unicode существуют еще несколько вариантов кодировки Unicode (UTF, Unicode Transformation Formats), основные из которых UTF-32, UTF-16, UTF-8 и UTF-7.

В кодировке UTF-32 каждая кодовая позиция представлена 32-разрядным двоичным числом. Это очень простая и очевидная система кодирования, хотя и неэффективная в плане разрядности кода. Кодировка UTF-32 используется редко, главным образом, в операциях обработки строк для внутреннего представления данных.

UTF-16 каждую кодовую позицию для символов из плоскости BMP представляет двумя байтами. Кодовые позиции из других плоскостей представлены так называемой суррогатной парой. Представление основных символов одинаковым числом байтов очень удобно в том плане, что можно прямо адресоваться к любому символу строки.

В кодировке UTF-8 коды символов меньшие, чем 128, представляются одним байтом. Все остальные коды формируются по более сложным правилам. В зависимости от символа его код может занимать от двух до шести байтов, причем старший бит каждого байта всегда имеет единичное значение. Иными словами, значение байта лежит в диапазоне от 128 до 255. Ноль в старшем бите байта означает, что код занимает один байт и совпадает по кодировке с ASCII. Схема формирования кодов UTF-8 показана в табл. 6.2

Таблица 6.2. Структура кодов UTF-8

Число байтов	Двоичное представление	Число свободных битов
1	0xxxxxxx	7
2	110xxxxx 10xxxxxx	11 (5 + 6)
3	1110xxxx 10xxxxxx 10xxxxxx	16 (4 + 6 × 2)
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21 (3 + 6 × 3)
5	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	26 (2 + 6 × 4)

В UTF-7 код символа также может занимать один или более байтов, однако в каждом из байтов значение не превышает 127 (старший бит байта содержит ноль). Многие символы кодируются одним байтом, и их кодировка совпадает с ASCII,

однако некоторые коды зарезервированы для использования в качестве преамбулы, характеризующей последующие байты многобайтового кода.

Стандарт Unicode обратно совместим с кодировкой ASCII, однако, если в ASCII для представления схожих по виду символов (минус, тире, знак переноса) применялся общий код, в Unicode каждый из этих символов имеет уникальную кодировку. Распределение кодов в Unicode иллюстрирует табл. 6.3.

Таблица 6.3. Блоки символов в стандарте Unicode

Коды	Символы
0–8191	Алфавиты — английский, европейские, фонетический, кириллица, армянский, иврит, арабский, эфиопский, бенгали, деванагари, гур, гуджарати, ория, телугу, тамильский, каннада, малайский, сингальский, грузинский, тибетский, тайский, лаосский, кхмерский, монгольский
8192–12287	Знаки пунктуации, математические операторы, технические символы, орнаменты и т. п.
12288–16383	Фонетические символы китайского, корейского и японского языков
16384–59391	Китайские, корейские, японские идеографы. Единый набор символов каллиграфии хань
59392–65024	Блок для частного использования
65025–65536	Блок обеспечения совместимости с программным обеспечением

Идентификаторы

Идентификатор, иначе *имя*, — это строка символов, используемая для идентификации некоторой сущности в программе и начинающаяся с буквы. В первых языках программирования применялись имена, состоящие только из одного символа. Это было естественно, так как ранние языки программирования были в основном математическими, а математики давно использовали односимвольные имена для формального обозначения неизвестных параметров.

Традиция была нарушена с появлением языка Fortran 1, где в именах разрешалось использовать до шести символов. Ограничение длины имени шестью символами сохранилось и в языке Fortran 77, но в языках Fortran 90 и C разрешенное количество символов в идентификаторах увеличилось до 31; в языке Ada вообще нет ограничения на длину идентификатора, причем все идентификаторы являются значащими. В некоторых языках, например C++, также нет ограничения на длину идентификатора, хотя иногда такое ограничение вводится создателями конкретных реализаций этих языков, чтобы таблица имен, в которой во время компиляции хранятся идентификаторы, была не слишком большой и сложной.

Общепринятым видом идентификатора считается строка с разумным ограничением длины, содержащая такие соединительные символы, как символ подчеркивания «_». Символ подчеркивания используется с той же целью, что и пробел в русских текстах, но при этом он не разрывает строку идентификатора, содержащую его. Большинство современных языков программирования позволяют использовать в идентификаторах соединительные символы.

В некоторых языках (C, C++ и Java) различаются прописные и строчные буквы; то есть идентификаторы в этих языках зависят от регистра. Например, в языке C++ три следующих идентификатора различны: `name`, `NAME` и `Name`. В некотором смысле это ухудшает читабельность, поскольку идентификаторы, внешне выглядящие очень похоже, на самом деле означают различные объекты. По сути зависимость от регистра нарушает принцип проектирования, гласящий, что языковые конструкции, имеющие одинаковый вид, должны иметь одинаковый смысл.

Разумеется, есть люди, которые не согласятся с утверждением, что зависимость от регистра является неудачным свойством идентификаторов. В языке C, например, зависимости от регистра можно избежать, используя исключительно идентификаторы, состоящие из строчных букв. Однако в языке Java такой зависимости не избежать, поскольку многие предопределенные идентификаторы содержат и прописные, и строчные буквы. Например, используемый в языке Java метод преобразования строки в целочисленное значение называется `parseInt`, а если написать его в виде `ParseInt` или `parseint`, то задание не будет распознано. Данная проблема относится к легкости написания программы, а не к ее читабельности, поскольку необходимость запоминать необычные правила написания слов затрудняет написание правильных программ.

В версиях языка Fortran вплоть до версии Fortran 90 в идентификаторах могли использоваться только прописные буквы, что было излишним ограничением. Многие реализации языка Fortran 77 разрешали использовать строчные буквы — просто они переводились в прописные для нужд компиляции.

Константы и литералы

Константа (постоянная величина) — это величина, значение которой остается неизменным во время выполнения программы. Различают целые, вещественные, символьные и строковые константы.

Целые константы содержат только цифры. Использование десятичной точки в целой константе не допускается. Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными.

Десятичная целая константа является последовательностью десятичных цифр, начинающейся не с нуля, если это не число нуль: 16, 32768, 0, 7. Отрицательными считают константы без знака, к которым применена операция изменения знака «-».

Восьмеричная целая константа всегда начинается с нуля, в ней запрещено появление цифр 8 и 9. Например, восьмеричная константа 016 имеет десятичное значение 14.

Шестнадцатеричная целая константа начинается с обязательной последовательности `0x` или `0X` и содержит одну или несколько шестнадцатеричных цифр (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

Вещественная константа представляет число в десятичной форме, например 3.1415927, 0.0000625, –10.2. У нее есть как целая часть, стоящая слева от десятичной точки, так и дробная часть, стоящая справа от нее. Вещественная константа отображает вещественное число, с помощью которого измеряются непрерывные величины: температура, расстояние, площадь. Вещественная константа, как правило, имеет ненулевую дробную часть.

Вещественная константа может включать следующие пять частей:

- целая часть (десятичная целая константа);
- десятичная точка;
- дробная часть (десятичная целая константа);
- символ экспоненты e или E;
- показатель десятичной степени (десятичная целая константа, возможно со знаком).

В записях вещественных констант могут опускаться:

- целая либо дробная часть (но не одновременно);
- десятичная точка или символ экспоненты с показателем степени (но не одновременно).

Примеры: 115.75, 1.5E–2, –0.025, .075, –0.85E2.

Забегая вперед, заметим: к записи вещественной константы может добавляться суффикс `F` (или `f`) либо `L` (или `l`). Суффикс `F` (`f`) говорит о том, что константе придают формат типа `float`, а суффикс `L` (`l`) означает формат типа `long double`.

Символьная константа представляет символ, который записывается в одиночных кавычках (апострофах): `'a'`, `'b'` и т. д. Значением символьной константы является числовой код символа. Когда компилятор встречает символьную константу, он заменяет ее на соответствующий ASCII-код (таблица ASCII-кодов предназначена для интерпретации символов как чисел). Например, константа `'a'` будет заменена числом 97.

Особую роль играют такие символьные константы, как управляющие последовательности (*escape-последовательности*). Они начинаются с символа «обратная косая черта». Название «управляющая последовательность» означает, что символ \ «управляет» интерпретацией следующих за ним символов последовательности. Так, `'\t'` воспринимается не как символ `'t'`, а как символ табуляции. Символ табуляции означает, что следующий символ будет напечатан через восемь позиций. Символ `'\n'` означает, что следующий вывод текста начнется с новой строки.

Управляющие последовательности можно использовать как в качестве отдельных констант, так и в составе строковых констант. Список управляющих последовательностей приведен в табл. 6.4.

Таблица 6.4. Управляющие последовательности

Управляющая последовательность	Символ
\a	Звуковой сигнал
\b	Пробел
\f	Перевод страницы
\n	Перевод в начало следующей строки
\r	Возврат каретки
\t	Табуляция
\\	Обратная косая черта
\'	Одинарные кавычки
\"	Двойные кавычки
\xdd	Шестнадцатеричный код символа
\?	Вопросительный знак

Поскольку при употреблении символьных и строковых констант символ \, а также одинарные и двойные кавычки по-особому интерпретируются компилятором, необходимо с помощью управляющих последовательностей обеспечить способ их включения в символьные и строковые константы в качестве обычных символов.

Если нам нужно, чтобы на экране компьютера появилась следующая фраза с кавычками:

```
"Все будет хорошо", сказала она
```

то в операторе вывода программы следует задать строку:

```
"\" Все будет хорошо \", сказала она."
```

Иногда хочется описать символьную константу, которая не может быть введена с клавиатуры, например псевдографический символ с ASCII-кодом 127. Чтобы сделать это, можно использовать управляющую последовательность вида `'\xdd'`, где `d` обозначает шестнадцатеричную цифру. Если вам требуется напечатать символ, представляющий собой закрашенный прямоугольник, то вы должны найти в ASCII-таблице код этого символа — 178. При переводе числа 178 в шестнадцатеричную систему счисления получим число B2. Таким образом, нужный символ представляется управляющей последовательностью `'\xB2'`.

Строковая константа — это последовательность символов, заключенная в кавычки (не в апострофы):

```
"Это строка, называемая строковой константой"
```

Повторим: среди символов строковой константы могут быть управляющие последовательности:

```
"У каждой эпохи свои причуды\n"
```

После вывода этой фразы на экран следующая фраза будет печататься на новой строке.

Литерал — это любая числовая, символьная или строковая константа, записанная непосредственно в тексте программы.

Например, в операторе присваивания из программы на языке C

```
y = 2.5 * x + 47;
```

содержится вещественный литерал `2.5` и целый литерал `47`.

Символы операций

Увы, естественные обозначения «`+`» и «`-`» применяются лишь для двух основных арифметических операций, но этим схожесть символов для одинаковых операций в разных языках и ограничивается. Например, операция «присвоить» обозначается в языке C как «`=`», а в языке Ada — как «`:=`». Операция «не равно» в языке C имеет обозначение «`!=`», в языке Pascal — «`<>`», в языке Ada — «`/=`». Все простейшие операции часто представляются или специальными символами (язык APL), или идентификаторами (язык LISP: `PLUS` для сложения, `TIMES` для умножения). Чаще всего упомянутые подходы комбинируются — некоторые операции обозначаются символами, а другие именуются идентификаторами, или специальными строками символов (в языке Fortran строка `.EQ.` обозначает операцию проверки на равенство).

Ключевые и зарезервированные слова

Ключевое слово — это идентификатор, используемый в качестве обязательного элемента синтаксиса какого-либо оператора или объявления. Приведем примеры: слово `if`, отмечающее начало условного оператора в языке C, или слово `for`, обозначающее начало оператора цикла; слово `type`, предваряющее объявление типа в языке Pascal. Ключевое слово называется *зарезервированным*, если синтаксис запрещает его использование в качестве идентификатора, определяемого программистом. Зарезервированные слова присутствуют в большинстве языков программирования, помогая транслятору обнаруживать синтаксические ошибки. Большинство операторов начинаются с зарезервированного слова, указывающего на категорию оператора: `read`, `case`, `while` и т. д.

Использование зарезервированных слов существенно упрощает синтаксический анализ в ходе трансляции. В языке Fortran, например, ситуация осложняется из-за того, что оператор, предваряемый словами `DO` или `IF`, может и не быть оператором цикла (или условным оператором). Поскольку `DO` и `IF` не считаются зарезервированными словами, программист имеет полное право использовать их в качестве имен переменных. В языке Cobol, напротив, зарезервированных слов слишком много, их трудно запомнить, поэтому возможно случайное употребление какого-либо из этих слов в качестве имени переменной. При добавлении новых зарезервированных слов в модифицированную версию языка тоже вероятны проблемы. Применение нового зарезервированного слова означает, что любая старая программа становится синтаксически неправильной, если в ней это слово используется в качестве имени переменной. В приводимых примерах программ мы будем стараться выделять зарезервированные слова полужирным шрифтом.

Во многих языках содержатся *предопределенные имена*, которые занимают некоторое среднее положение между зарезервированными словами и именами, определяемыми пользователем. С одной стороны, их смысл предопределен. С другой стороны, они могут переопределяться пользователем. Например, такие встроенные в язык Ada имена типов данных, как `Integer` и `Float`, предопределены. Эти имена не зарезервированы — их переопределение доступно любой программе на языке Ada. В языке Pascal обычные имена подпрограмм ввода-вывода, в том числе и названия подпрограмм `readln` и `writeln`, предопределены.

Определения предопределенных имен в языках Pascal и Ada должны быть видимы компиляторами этих языков, поскольку проверка типов здесь производится в период компиляции. В обоих языках перечисленные выше примеры предопределенных имен видимы компилятором неявно. В языке Ada предопределенные имена стандартных процедур ввода и вывода `get` и `put` видимы. Их видимость обеспечивается оператором `with`, который записывается программистом.

В языках C и C++ многие имена предопределены в стандартных библиотеках. Например, имена функций ввода и вывода языка C `printf` и `scanf` описаны в библиотеке `stdio`. Доступ компилятора к именам из библиотек обеспечивается соответствующими заголовочными файлами.

Необязательные слова

Необязательными являются слова, которые вставляются в операторы для удобства чтения программы. Множество таких слов предусматривается в языках Cobol, Ada. Например, в Cobol оператор **GO** `TO метка` содержит обязательное ключевое слово **GO** и необязательное слово `TO`, облегчающее чтение программы. В закрывающем слове языка Ada для типа запись

```
end record Date;
```

имя `Date` тоже лишь облегчает идентификацию конкретного типа.

Комментарии

Комментарии в программе считаются важной частью ее документирования. Языки программирования предусматривают несколько способов включения комментариев:

1. Специальная строка комментария. В программе на Basic строка комментария начинается с ключевого слова `REM`.
2. Произвольное место программы, в котором текст комментария заключен в ограничивающие маркеры. В языке C используют маркеры `/*` и `*/`, а в языке Pascal — маркеры `(*` и `*)`.
3. Произвольное место программы, в котором текст комментария продолжается только до конца строки. В языке Ada начало такого комментария обозначается двумя дефисами `- -`, а в C++ — парой символов `//`.

Слабым местом второго из этих способов является то, что если пропущен символ конца комментария, весь последующий текст программы воспринимается как непрерывный комментарий. Ограничением этой «непрерывности» станет конец следующего комментария. Следовательно, фрагмент программы не будет обработан транслятором, поскольку эту синтаксическую ошибку транслятор не заметит.

Пробелы

В разных языках применяются самые разнообразные правила использования пробелов. В языке C пробелы не являются значащими символами (за исключением их записи в строковых литералах). Существуют языки, где пробелы считаются разделителями и играют важную синтаксическую роль. В языке Snobol4 пробел, с одной стороны, обозначает операцию конкатенации, а с другой стороны, считается разделителем между элементами в операторах. Это приводит к большой путанице. На ранней стадии развития языка C пара символов «=+» означала единую операцию, а строка = + соответствовала последовательности из двух операций. Теперь определение языка исправлено, единичная операция получила обозначение +=, в то время как запись + = приводит к синтаксической ошибке.

Разделители и скобки

Разделитель — это элемент, обозначающий конец некоторой синтаксической конструкции (оператора или выражения). Чаще всего разделителем является точка

с запятой (;). Скобки считаются парой разделителей, например *круглые скобки*, *фигурные скобки*, или пары вида `begin … end`. Разделители решают очень серьезную задачу. Они устраняют неоднозначность посредством явного определения границ конкретной синтаксической конструкции.

Выражения

Выражение отображает формулу, в которой обрабатываются элементы данных и вычисляется некоторый результат. Выражения являются основными строительными компонентами для создания операторов (в императивных языках). В функциональных языках выражения задают поток управления в программе. Содержательно мы обсудили выражения в главе 4. В текущей главе мы будем рассматривать синтаксис выражений.

Операторы

Операторы — наиболее важные элементы императивных языков. Синтаксис операторов оказывает решающее воздействие на регулярность языка, удобство чтения и создания программы. В одних языках принята единая форма для записи операторов, в других различные виды операторов имеют разные формы типов. В первом случае говорят о максимальной регулярности языков, во втором — об удобстве чтения и написания программ. В языке Snobol4 определен синтаксис одного оператора — оператора сопоставления с образцом и подстановки, все остальные разновидности операторов получаются из него посредством удаления некоторых элементов. Операторы императивных языков подробно рассматривались в главе 5. В текущей главе мы обратимся к синтаксическим характеристикам операторов.

Лексемы и лексический синтаксис

С одной стороны, реальный язык программирования описывается достаточно сложными синтаксическими правилами. Синтаксические правила определяют, какие именно строки имеют право на существование в языке.

С другой стороны, совершенно естественным является стремление упростить синтаксические правила. Для простоты в формальные описания синтаксиса обычно не включают описания синтаксических единиц самого нижнего уровня (атомов). Подобные атомы группируют в обобщенные единицы — *лексемы*. Лексемы считают основным инструментом лаконичной записи синтаксических правил.

Работу с лексемами, их детализацию описывает лексический синтаксис языка. *Лексический синтаксис* задает соответствие между реальной программой и синтаксическими лексемами.

Действительную форму представления (вхождения) лексемы в реальной программе называют *спеллингом*. Спеллинги можно считать значениями лексем. Обычно спеллинги показывают в виде индексов при лексемах.

Положим, что синтаксис языка задает следующие лексемы: `name` (для имен), `number` (для чисел) и `op` (для операций).

Правила лексического синтаксиса, указывающие возможные значения (спеллинги) лексем сведены в табл. 6.5. Здесь возможные значения лексемы отделены друг от друга запятыми.

Таблица 6.5. Лексический синтаксис языка

Лексема	Спеллинги (значения) лексемы
`name`	a, b, c, d, e, x, y, z, ...
`number`	0, 1, 2, 3, 4, 5, 6, 7, ...
`op`	+, –, *, ...

Тогда реальной программной строке

```
x = 5 * a + 7
```

будет соответствовать следующая лексическая структура

```
name op number op name op number
```

а с учетом значений (спеллингов) лексем можно записать ее конкретизацию:

$\texttt{name}_{x}\ \texttt{op}_{=}\ \texttt{number}_{5}\ \texttt{op}_{*}\ \texttt{name}_{a}\ \texttt{op}_{+}\ \texttt{number}_{7}$

где спеллинги показаны в виде индексов при лексемах.

Согласитесь, что правило описания лексической структуры имеет максимально обобщенный вид и описывает целое семейство программных строк. В этих строках можно менять имена переменных, значения констант и значки операций. Нужно лишь оставаться в рамках ограничений, указываемых лексическим синтаксисом. В частности, приведенная выше лексическая структура полностью соответствует следующим реализациям программных строк:

```
y = 2 * b + 5
a = 3 + c - 4
z = 4 + e * 6
```

При изменении состава синтаксических лексем меняется и форма лексической структуры. Например, если из синтаксиса удалить лексему для операций, то запись программной строки

```
a * b + 4 * c / d
```

получит следующую лексическую структуру

$\texttt{name}_{a}\ *\ \texttt{name}_{b}\ +\ \texttt{number}_{4}\ *\ \texttt{name}_{c}\ /\ \texttt{name}_{d}$

Между лексемами обычно вставляются пробелы, которые не изменяют смысл программы.

Абстрактный синтаксис и абстрактные синтаксические деревья

Для повышения наглядности условимся синтаксические структуры изображать графически. Наглядным графическим отображением структуры выражения является дерево, вершины которого помечаются именами операций и операндов, а ребра (линии) показывают отношения «родитель-потомок» между вершинами. Деревья обычно рисуются сверху вниз, так что родители располагаются выше потомков.

Абстрактный синтаксис языка определяет значащие компоненты каждой конструкции языка. Префиксное выражение + a b, инфиксное выражение a + b и постфиксное выражение a b + имеют одинаковые значащие компоненты: операцию + и подвыражения (операнды) a и b. Абстрактный синтаксис этого выражения представляется деревом, изображенным на рис. 6.1.

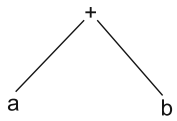


Рис. 6.1. Дерево выражения a + b

Если заранее известен абстрактный синтаксис языка, то возможен выбор лучшей грамматики как средства формального и точного определения синтаксиса.

Абстрактное синтаксическое дерево (АСД) — это узел с k ($k \geq 0$) деревьями-потомками. Если $k = 0$, то дерево состоит из одного узла, без деревьев-потомков. Узел без потомка называется листом. Корень дерева — это узел без родителя, то есть узел, который не является потомком другого узла.

Если выражение формируется применением операции op к операндам E_1, E_2,..., E_k (для $k \geq 0$), то его абстрактное синтаксическое дерево включает узел для op с k потомками — деревьями для подвыражений. Диаграмма дерева показана на рис. 6.2.

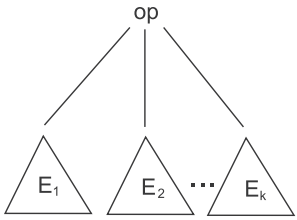


Рис. 6.2. Общий вид абстрактного синтаксического дерева

Листья этого дерева изображены с помощью треугольников, чтобы подчеркнуть, что они могут иметь внутреннюю структуру. При необходимости внутренняя структура детализируется отдельно. Если выражение является константой или переменной, то его дерево состоит из одного листа.

В качестве примера на рис. 6.3 представлено дерево для выражения b * b – 4 * a * c.

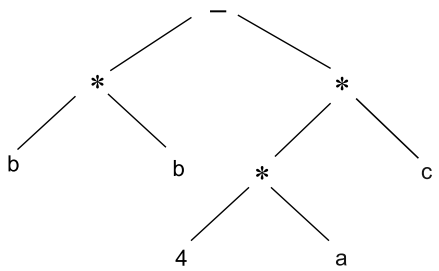


Рис. 6.3. Дерево выражения b * b – 4 * a * c

Как строится это дерево? Прежде всего выделяется операция выражения, которая выполняется последней (здесь это минус). Рисуется дерево для треугольника `E1–E2`, где `E1` это `b * b`, а `E2` — это `4 * a * c`. В этом дереве имеется корень для «–», двумя потомками корня являются поддеревья для `E1` и `E2`. Далее последовательно детализируются поддеревья `E1` и `E2`. Например, для `E2` подвыражение `4 * a * c` представляется как `E3 * E4`, где `E3` — это `4 * a`, а `E4` — это `c`. Следовательно, соответствующее поддерево имеет узел для * с поддеревьями-потомками для `4 * a` и `c`. Процесс «наращивания» дерева прекращается, когда все листья становятся неделимыми элементами.

Абстрактное синтаксическое дерево отображает структуру из операций и операндов выражения. Эта структура не зависит от нотации, используемой для записи выражения.

Приведенное выше дерево отражает абстрактный синтаксис каждого из следующих выражений:

Префиксная форма	– * b b * * 4 a c
Инфиксная форма	b * b – 4 * a * c
Постфиксная форма	b b * 4 a * c * –

Для записи конкретной нотации достаточно применить (рекурсивно) к дереву одно из трех правил обхода (рис. 6.4).

Рис. 6.4. Правила обхода абстрактных синтаксических деревьев

Как видим, предлагаются следующие маршруты обхода:

- Корень — левый потомок — правый потомок (результат: префиксная форма).
- Левый потомок — правый потомок — корень (результат: постфиксная форма).
- Левый потомок — корень — правый потомок (результат: инфиксная форма).

С помощью АСД фиксируется абстрактный синтаксис языка.

Абстрактные синтаксические деревья могут применяться и к другим конструкциям путем формирования для конструкций подходящих операций.

Например, для выражения `if a > b then a else b` можно построить дерево созданием операции `if-then-else` (рис. 6.5).

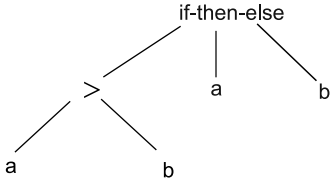


Рис. 6.5. Абстрактное синтаксическое дерево выражения if a > b then a else b

Грамматики в языках программирования

Внешний вид элементов языка программирования задают с помощью конкретного синтаксиса. Он описывает такие лексические детали, как размещение ключевых слов и знаков пунктуации.

Для спецификации конкретного синтаксиса применяют грамматики.

Грамматика языка вводит иерархическую структуру, называемую деревом разбора над программами языка. Положим, что мы имеем язык вещественных чисел. Тогда дерево разбора для строки 3.14 принимает вид, изображенный на рис. 6.6.

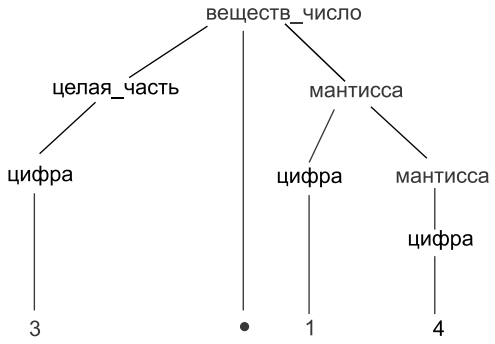


Рис. 6.6. Дерево разбора для строки 3.14

Листья внизу дерева разбора помечаются терминалами или лексемами (например: `3`); лексемы представляют сами себя. Напротив, промежуточные узлы дерева разбора помечаются нетерминалами (например: `веществ_число`, `цифра`). Нетерминалы представляют языковые конструкции. Каждый узел в дереве разбора основывается на правиле подстановки. Правило подстановки определяет нетерминал с помощью последовательности терминалов и нетерминалов. Корень дерева разбора для строки 3.14 основывается на следующем неформальном правиле подстановки.

Вещественное число состоит из целой части, точки и дробной части.

Грамматику языка образуют все лексемы, нетерминалы, правила подстановки и особый нетерминал, называемый стартовым нетерминалом. Стартовый нетерминал может представлять часть полной программы (при изучении фрагментов ЯП).

Все лексемы и нетерминалы называют символами грамматики (или просто символами).

Контекстно-свободная грамматика

Повторим, что для точного и формального задания синтаксиса языка используют грамматику. В середине 50-х годов лингвист Хомский определил четыре вида грамматик [57]. Два вида оказались полезны для описания синтаксиса ЯП (контекстно-свободная и регулярная грамматики).

Контекстно-свободная грамматика точно определяет конкретный синтаксис и включает в себя:

1. *Набор терминалов*. Они являются атомарными символами языка.
2. *Набор нетерминалов*. Это «сборки», представляющие конструкции языка.
3. *Набор правил подстановки*, задающих компонентный состав конструкций. Каждое правило подстановки имеет левую и правую части, разделяемые символом «может быть». В левой части находится нетерминал, в правой части — строка из набора терминалов и нетерминалов.
4. *Стартовый нетерминал*. Он представляет главную конструкцию языка.

ПРИМЕЧАНИЕ

Правила подстановки для стартового нетерминала всегда указываются первыми.

Приведем примеры элементов контекстно-свободной грамматики:

1. **Терминалы**

```
name
0 .. 9
:=
+
```

2. **Нетерминалы**

```
условный_оператор
оператор_while
expression
присваивание
```

3. **Правило подстановки**

```
присваивание может быть name := a + expression
```

4. **Стартовый нетерминал**

```
program
```

Форма Бэкуса—Наура (BNF)

Понятие контекстно-свободной грамматики не зависит от нотации, которая используется для ее записи. Обычно такие нотации называют *метаязыками* (метаязык — язык для описания другого языка).

Форма Бэкуса—Наура (BNF) — одна из таких нотаций. BNF стала популярной после ее использования для представления отчета по языку программирования Алгол 60.

Охарактеризуем выразительные средства (метасимволы) этой формы [50].

1. **Терминалы и нетерминалы.** Нетерминалы заключаются в угловые скобки < и >, пустая строка записывается в виде <empty>. Терминалы могут заключаться в кавычки.

   Например, в BNF-грамматике для вещественных чисел нетерминалами являются <веществ_число>, <целая_часть>, <дробн_часть> и <цифра>. Лексемами (терминалами) считаются цифры 0, 1,...9 и десятичная точка «.».

2. **Правила подстановки.** Метасимвол ::= означает «может быть», метасимвол | означает «или». Метасимволы | отделяют друг от друга альтернативы правых частей (правил подстановки). Например, правило подстановки

```
<дробн_часть> ::= <цифра>|<цифра> <дробн_часть>
```

должно быть прочитано так:

```
"Дробная часть может быть цифрой или может быть цифрой,
        за которой следует дробная часть".
```

Например, дробная часть может быть единственной цифрой `2` или может быть цифрой, за которой следует другая часть, как в `142`, где за цифрой `1` следует `42`.

3. Каждая **альтернатива**, отделяемая символом | , соответствует самостоятельному правилу, так что вышеуказанное правило можно переписать в виде:

```
<дробн_часть> ::= <цифра>
<дробн_часть> ::= <цифра> <дробн_часть>
```

4. **Необязательную конструкцию** удобно задавать с помощью переменной `<empty>` (пустой строки с нулевой длиной).

```
<целая_часть> ::= <empty > |<послед-ть_цифр>
```

Например, допустим, что `0` в `0.5` является необязательным элементом.

Такой синтаксис определяется следующими правилами подстановки:

```
<веществ_число> ::= <целая_часть> . <дробн_часть>
<целая_часть> ::= <empty > | <послед-ть_цифр>
<дробн_часть> ::= <послед-ть_цифр>
```

Здесь задано, что вещественное число имеет целую часть, десятичную точку и дробную часть. Целая часть является необязательной последовательностью цифр, дробная часть — просто последовательность цифр. Это следует из правила

```
<целая_часть> :: = <empty >
```

То есть целая часть может быть пустой, например допустима строка «.5», где дробная часть является последовательностью из одной цифры.

Деревья разбора

Правила подстановки грамматики являются правилами для создания программных строк.

Пример строки (правда, совсем не программной):

```
Мама мыла раму
```

Нам будет удобно использовать эту строку для иллюстрации особенностей построения правил подстановки.

Очень важным приложением правил подстановки является также проверка правильности строк, созданных программистом, — грамматический разбор строк. При решении этой задачи целесообразно применять специальный инструмент — дерево разбора [1, 31, 32, 49, 86].

Дерево разбора иллюстрирует процесс построения строки, имеет следующие характеристики:

1. Каждый лист помечается терминалом или `<empty>`.
2. Каждый неконечный узел помечается нетерминалом.

3. Метка неконечного узла является левой частью некоторого правила подстановки. Метки потомков узла (от левого к правому) формируют правую часть этого правила подстановки.
4. Корень помечается стартовым нетерминалом.

Создание дерева разбора начинается с корня. На каждом шаге строительства количество узлов дерева растет в результате применения конкретного правила подстановки. Правило применяется к одному из узлов дерева. Итогом применения правила является появление потомков у этого узла.

Дерево разбора генерирует строку, формируемую чтением терминалов у ее листьев. Терминалы читаются слева направо. Строка разрешается языком, если (и только если) она генерируется некоторым деревом разбора.

Конструирование дерева разбора называется *грамматическим разбором*.

Пример: BNF-правила для вещественных чисел

```
<веществ_число> ::= <целая_часть> . <дробн_часть>
<целая_часть> ::=  <цифра> |<целая_часть> <цифра>
<дробн_часть> ::= <цифра> |<цифра> <дробн_часть>
<цифра> ::=  0 |1 |2 |3 |4 |5 |6 |7 |8 |9
```

Дерево разбора любого числа для этой грамматики начинается с корня `<веществ_число>`, к которому добавляются три потомка (рис. 6.7). Это соответствует применению первого правила подстановки.

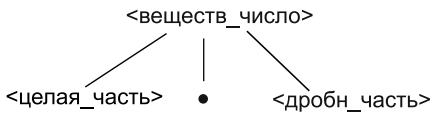


Рис. 6.7. Первый шаг создания дерева разбора

В дальнейшем это дерево может расти за счет добавления потомков к нетерминалам `<целая_часть>`, `<дробн_часть>`.

Например, для вещественного числа `123.789` финальное дерево разбора примет вид, показанный на рис. 6.8.

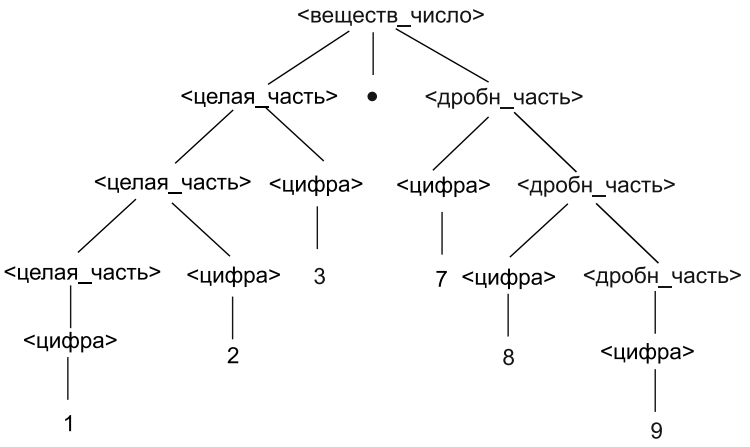


Рис. 6.8. Финальное дерево разбора

Чтобы прочесть результат разбора, надо обойти листья этого дерева, двигаясь слева направо. Такой обход дает нам строку:

```
123.789
```

Получив этот результат, делаем вывод: строка записана правильно, в соответствии с правилами заданной грамматики.

Деревья разбора для реальных программ достаточно велики, поэтому работу по конструированию деревьев разбора лучше выполнять с помощью компьютера.

Синтаксическая неоднозначность

Грамматика языка программирования является синтаксически неоднозначной, если некоторая строка в его языке имеет более одного дерева разбора. Языки программирования могут описываться неоднозначными грамматиками. Если неоднозначности существуют, вводятся соглашения, приводящие к одному дереву разбора для каждой строки.

Например, следующая грамматика

```
<E> ::=  <E> - <E>|0|1
```

неоднозначна, так как строка `1 – 0 – 1` имеет два дерева разбора, соответствующие двум вариантам `(1 – 0) – 1` и `1 – (0 – 1)`. Возможны два дерева разбора этой строки (рис. 6.9).

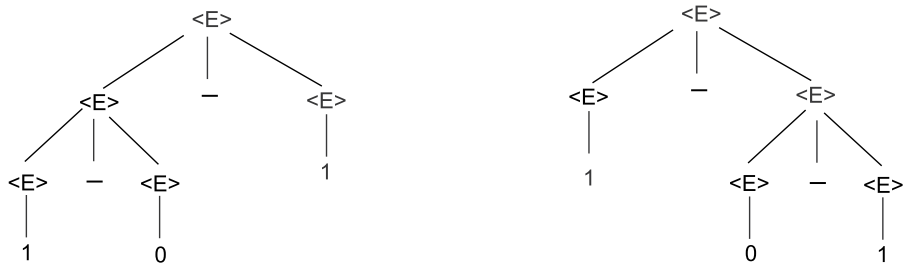


Рис. 6.9. Два дерева разбора строки 1 – 0 – 1

Неоднозначность повисшего Else

Известный пример — неоднозначность повисшего Else. Она возникает, если в грамматике есть два правила подстановки:

```
<S> ::=  if <E> then <S>
<S> ::=  if <E> then <S> else <S>
```

где `<S>` — операторы, `<E>` — выражения.

Ни одно правило само по себе не приводит к неоднозначности. Вместе, тем не менее, они разрешают следующие конструкции:

```
if <E1> then if <E2> then <S1> else <S2>
```

в которых неясно, к какому `if` относится `else`.

Существует два варианта деревьев разбора (рис. 6.10).

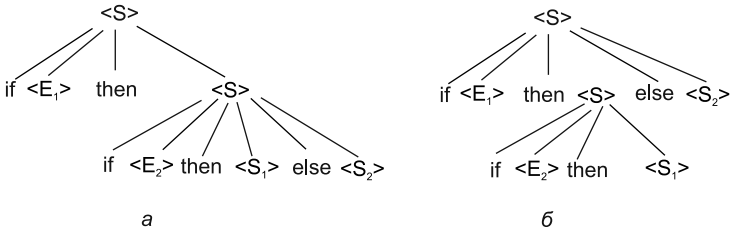


Рис. 6.10. Два дерева разбора строки if <E1> then if <E2> then <S1> else <S2>

В варианте (*а*) `else` относится к ближайшему `if`, в варианте (*б*) — к дальнему `if`. Обычно принято сопоставлять `else` ближайшему свободному `if`.

Выводы — линейная форма грамматического разбора

Различают два способа разбора — нисходящий и восходящий. При нисходящем разборе дерево разбора проходится от корня к листьям, при восходящем разборе — от листьев к корню. Обычно используется нисходящий разбор.

Рассмотрим моментальные снимки деревьев нисходящего разбора вещественного числа (рис. 6.11).

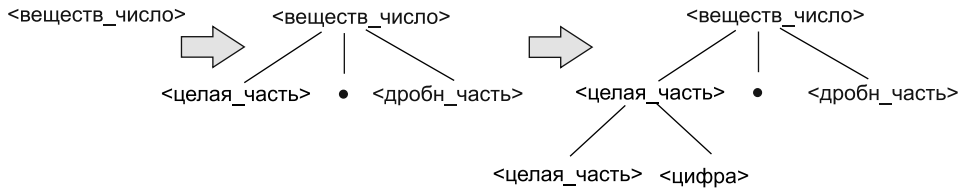


Рис. 6.11. Моментальные снимки дерева разбора

Здесь представлены три момента времени в строительстве дерева разбора. В первый момент дерево состоит из одной вершины. Во второй момент к дереву добавлены три вершины и т. д.

Если рассматривать только листья и игнорировать оставшуюся часть текущего дерева, мы получим последовательность, называемую выводом:

```
<вещ_число> ⇒ <цел_часть>.<дробн_часть> ⇒ <цел_часть><цифра>.<дробн_часть>
```

Здесь стрелка ⇒ обозначает отношение следования.

Вывод состоит из последовательности строк, начинающихся со стартового нетерминала. Каждая строка получается замещением нетерминала правой частью одного из его правил подстановки. Вывод заканчивается, когда получают строку, целиком состоящую из терминалов.

Пример вывода для числа 21.89

```
<вещ_число> ⇒ <цел_часть> . <дробн_часть>
            ⇒ <цел_часть> <цифра> .<дробн_часть>
```

```
⇒ <цифра> <цифра> . <дробн_часть>
⇒ 2 <цифра> . <дробн_часть>
⇒ 21 . <дробн_часть>
⇒ 21 . <цифра> <дробн_часть>
⇒ 21 . 8 <дробн_часть>
⇒ 21 . 8 <цифра>
⇒ 21 . 89
```

Наиболее популярен *левосторонний вывод* — в нем каждый раз замещается самый *левый* нетерминал.

Чем удобен механизм вывода? Ответ прост: не надо рисовать картинки, отображающие дерево разбора.

Списки в инфиксных выражениях

При написании правил подстановки для конструирования выражений очень важно обеспечить рост длины выражения, управление ростом, завершение роста.

Для решения этих вопросов удобно использовать понятие списка. Выражение `a + b + c` можно рассматривать как список элементов, отделяемых (или склеиваемых) символом +, элементы списка называются термами. Термы обычно атомарны, но могут и иметь внутреннюю структуру.

Например, строку

```
Мама мыла раму
```

можно представить как *аддитивный список* из термов-символов:

```
М+а+м+а+  +м+ы+л+а+  +р+а+м+у
```

Идея списка применима и к термам-структурам. Терм `4 * a * c` может сам рассматриваться как список элементов, отделенных символами *, элементы называются *факторами* (рис. 6.12). Обычно список из факторов называют *мультипликативным списком*.

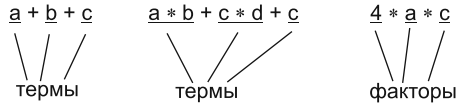


Рис. 6.12. Термы и факторы

Рассмотрим грамматику для арифметических выражений:

```
<E> ::=  <E> + <T>|<E> - <T>|<T>
<T> ::=  <T> * <F>|<T> / <F>|<F>
<F> ::=  number|name|(<E>)
```

Здесь нетерминалы `<E>`, `<T>`, `<F>` представляют выражения, термы и факторы соответственно. Применим подход, ориентированный на списки.

Для генерации аддитивного списка термов в деревьях разбора используем правила подстановки

```
<E> ::=  <E> + <T>
        |<E> - <T>
```

Каждый раз, когда применяется одно из правил подстановки, к списку добавляется очередной терм (рис. 6.13).

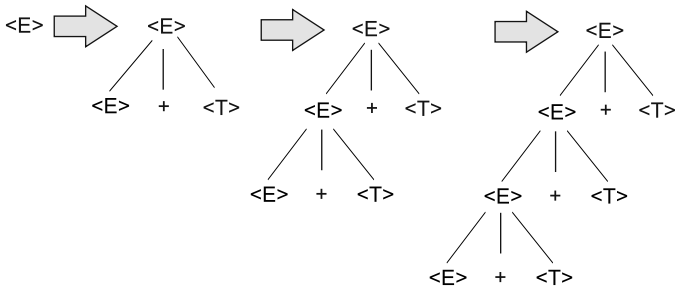


Рис. 6.13. Шаги наращивания аддитивного списка

Список перестает расти при применении правила подстановки

```
<E> ::= <T>
```

Подобное правило часто называют заглушкой, поскольку оно запрещает дальнейший рост списка.

Эти правила подстановки создают аддитивный список термов, отделяемых или символами +, или символами –.

Правила подстановки для `<T>` похожи на правила для `<E>` и приводят к созданию мультипликативных списков.

На рис. 6.14 показано дерево разбора для строки, включающей в себя один аддитивный и два мультипликативных списка:

$number_7 * number_7 - number_4 * number_2 * number_3$

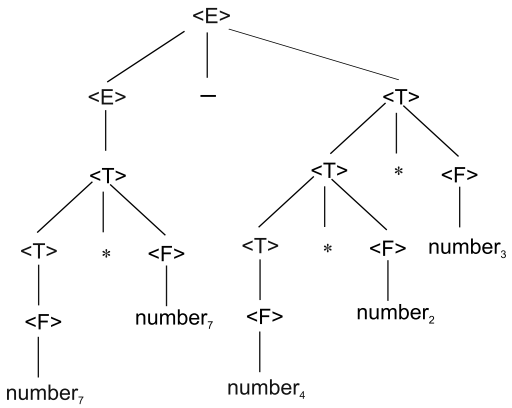


Рис. 6.14. Дерево разбора на основе аддитивно-мультипликативного списка

Для получения финального дерева пришлось выполнить 12 шагов строительства в терминах правил подстановки заданной грамматики.

Переход к конкретному синтаксису

В общем случае различают несколько уровней описания синтаксиса языка программирования:

- абстрактный синтаксис;
- лексический синтаксис;
- конкретный синтаксис.

Многоуровневая иерархия позволяет учесть специфику такой сложной системы, как язык программирования, обеспечивает концентрацию внимания на всех важных вопросах. Абстрактный синтаксис нацелен на архитектуру выражений любой сложности, лексический синтаксис описывает внутреннюю структуру, значения таких строительных элементов, как лексемы. Конкретный синтаксис включает в себя средства для развернутого описания всех синтаксических элементов, вбирая всю информацию абстрактного и лексического уровней.

Естественно, что грамматика конкретного синтаксиса должна быть согласована с абстрактным синтаксисом. Это означает: правила подстановки выбирают так, чтобы деревья разбора были как можно ближе к абстрактным синтаксическим деревьям.

Условимся, что выражения записываются в инфиксной форме. Тогда следует требовать, чтобы обработка ассоциативности операций в абстрактном и конкретном синтаксисах происходила одинаково.

Например, для выражений с левоассоциативными операциями их абстрактные синтаксические деревья растут влево и вниз. Для выражения `4 – 2 – 1` такое дерево изображено на рис. 6.15.

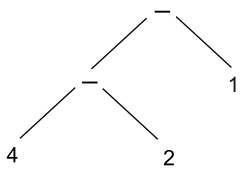


Рис. 6.15. Абстрактное синтаксическое дерево строки 4 – 2 – 1

Оно растет вниз и влево, соответствуя группировке `(4 – 2) – 1`.

Как показано на рис. 6.16, такой же рост дерева разбора (инструмента конкретного синтаксиса) обеспечивают правила подстановки конкретной *L*-грамматики:

```
<L>  ::=  <L> + number
        | <L> – number
        |  number
```

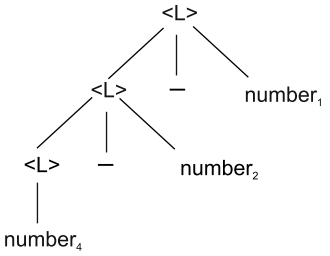


Рис. 6.16. Дерево разбора строки 4 – 2 – 1 средствами L-грамматики

Видим, что здесь стартовым нетерминалом грамматики является `<L>`, а первые два правила подстановки обеспечивают рост строки влево за счет *левой рекурсив-*

ности. В них нетерминал из левой части правила повторяется в левой позиции правой части.

Соответственно третье правило `<L> ::= number` служит для прекращения роста строки.

Проиллюстрируем механизм левой рекурсивности простейшим примером. Здесь необходимая строка создается за три шага:

```
Направление роста строки ←
Шаг 1:                              p+a+m+y
Шаг 2:                  м+ы+л+а+  +p+a+m+y
Шаг 3:     M+a+m+a+  +м+ы+л+а+  +p+a+m+y
```

В свою очередь, для выражений с правоассоциативными операциями их абстрактные синтаксические деревья растут вправо и вниз. Для строки `x := y := z` такое дерево представлено на рис. 6.17.

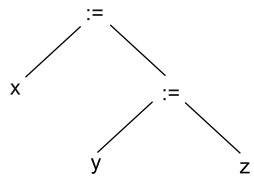


Рис. 6.17. Абстрактное синтаксическое дерево строки x := y := z

Оно растет вниз и вправо и соответствует группировке `x := (y := z)`.

Как показано на рис. 6.18, правый рост дерева разбора обеспечивают правила подстановки конкретной *R*-грамматики

```
<R>  ::=  name := <R>
        | name
```

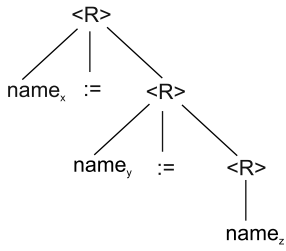


Рис. 6.18. Дерево разбора строки x := y := z средствами R-грамматики

Видим, что здесь стартовым нетерминалом грамматики является `<R>`, а первое правило подстановки обеспечивает рост строки вправо за счет *правой рекурсивности*. В нем нетерминал из левой части правила повторяется в правой позиции правой части.

Соответственно второе правило `<R> ::= name` служит для прекращения роста выражения.

Приведем пример по правой рекурсивности. Необходимая строка создается за три шага:

```
Направление роста строки →
Шаг 1:     M+a+m+a
```

```
Шаг 2:      М+а+м+а+  +м+ы+л+а
Шаг 3:      М+а+м+а+  +м+ы+л+а+  +р+а+м+у
```

Краткие выводы

1. Обе грамматики однозначны.
2. *L*-грамматика более пригодна для *левоассоциативных* операций, так как ее деревья разбора растут вниз и влево, поэтому они ближе к абстрактному синтаксису выражений.
3. *R*-грамматика более пригодна для *правоассоциативных* операций, поскольку ее деревья разбора растут вниз и вправо, поэтому они ближе к абстрактному синтаксису выражений.

Обработка ассоциативности и приоритетности

В этом разделе речь пойдет о конструировании правил подстановки для конкретной грамматики языка программирования. Как правило, исходными данными для решения такой задачи являются сведения о встроенных операциях языка.

Синтаксис выражений языка можно охарактеризовать таблицей, указывающей ассоциативность и приоритетность операций. Для языка С она сведена в табл. 6.6.

Таблица 6.6. Операции языка С

Обозначения	Название	Ассоциативность
=	присваивание	правая
||	логическое или	левая
& &	логическое и	левая
|	побитовое или	левая
^	побитовое исключающее или	левая
&	побитовое и	левая
= = , !=	эквивалентность, неэквивалентность	левая
< , <=, >, >=	отношения	левая
<< , >>	сдвиги	левая
+ , –	аддитивные операции	левая
*, / , %	мультипликативные операции	левая

Все операции на одной строке имеют одинаковую ассоциативность и приоритет. Операция присваивания является *правоассоциативной*, все остальные операции — *левоассоциативные*. Первая строка таблицы характеризуется минимальным приоритетом операций, а последняя строка — максимальным.

Грамматика на основе таблицы операций создается в следующем порядке:

1. Для каждой строки таблицы операций (каждого уровня приоритета) выбирается *нетерминал*.

2. Для факторов (или наименьших подвыражений) создается *дополнительный нетерминал*.
3. Строки таблицы обрабатываются последовательно: начинают с первой, а заканчивают последней строкой. Для *левоассоциативных* операций строятся леворекурсивные правила подстановки, для *правоассоциативных* операций — праворекурсивные правила подстановки.

   Например, рассмотрим таблицу с тремя уровнями приоритета (табл. 6.7).

Таблица 6.7. Операции проектируемой грамматики языка

Операции	Ассоциативность	Нетерминал
: =	правая	<A>
+ , –	левая	<E>
*, /	левая	<T>
		<F>

Пусть нетерминалами трех уровней будут `<A>` (от слова Assignment), `<E>` (от слова Expression), `<T>` (от слова Term), а нетерминалом для факторов — `<F>` (от слова Factor).

Обратим внимание, что придуманные нами нетерминалы автоматически получают уровни приоритетов тех операций, которые они «обслуживают». А это значит, что нетерминал `<A>` имеет самый малый, а нетерминал `<F>` — самый большой приоритет.

Порядок следования правил подстановки выбирается с определенным смыслом. Он реализует идею: в низкоприоритетный список всегда можно вкладывать список с более высоким приоритетом (имеется в виду приоритет операций, «склеивающих» операнды — элементы списка). Это значит, что в список из выражений можно вкладывать список из термов, а в список из термов — список из факторов.

При проектировании правила подстановки для операции приходится решать вопрос об именах операндов. Если правило рекурсивно, имя одного из операндов известно: им является нетерминал, обслуживающий строку, в которой размещается символ операции. Например, для операций + и – одним из операндов будет `<E>`. А вот откуда взять имя другого операнда? Логика подсказывает, что им может быть нетерминал для операции, приоритет которой на единицу больше приоритета текущей операции. (Вспомните идею вложения списков!) Следовательно, для операций + и – другим операндом должен стать `<T>`, а для умножения и деления — нетерминал `<F>`.

После выбора имен операндов нужно выбрать, какой из операндов будет левым, а какой — правым. Ответ на этот вопрос дает ассоциативность операции.

Так как присваивание является правоассоциативной операцией, то правило подстановки для `<A>` соответствует *R*-грамматике и должно быть праворекурсивным (повторяющееся имя становится правым операндом).

Правила подстановки для `<E>` и `<T>` соответствуют *L*-грамматике и должны использовать левую рекурсию (повторяющееся имя становится левым операндом).

Таким образом, грамматика для нашей таблицы операций принимает вид:

```
<A>  ::=  <E> := <A> | <E>
<E>  ::=  <E> + <T> | <E> - <T> | <T>
<T>  ::=  <T> * <F> | <T> / <F> | <F>
<F>  ::=  (<E>) | name | number
```

В каждой группе правил предусмотрено правило-заглушка, дающая запрет на рост соответствующего списка. Отметим, что последняя группа правил разрешает использовать в качестве факторов два вида лексем.

Правило `<F> ::= (<E>)` играет особую роль. Оно позволяет нарушить естественный механизм вложения списков и обеспечивает возможность создания сложных иерархических структур: разрешает вкладывать в факторы списки из термов, в которые, в свою очередь, могут вкладываться списки из факторов.

Проиллюстрируем эту возможность следующим примером. Пусть исходная строка из факторов имеет вид

```
a * b * c
```

Положим, что фактор `a` нужно заменить списком термов

```
x + y + z
```

В результате такой замены можно получить строку

```
(x + y + z) * b * c
```

Правило `<F> ::= (<E>)` позволяет сделать это.

Расширенная BNF

Несмотря на элегантность и мощность BNF-грамматики, она не является идеальным средством для передачи правил конкретного языка программирования программисту-практику. Основная причина такой «неидеальности» заключается в простоте правил BNF, которая приводит к достаточно громоздкому представлению синтаксиса необязательных, альтернативных и повторяющихся элементов. Например, чтобы выразить простую мысль о том, что «целое со знаком» является последовательностью цифр, начинающейся с необязательного символа плюс или минус, в BNF приходится создавать сложный ряд рекурсивных правил:

```
<целое со знаком> ::= +<целое> | -<целое>
<целое> ::= <цифра> | <целое><цифра>
```

Рассмотрим некоторые расширения BNF, упрощающие запись типовых синтаксических ситуаций.

Для записи контекстно-свободной грамматики могут использоваться различные метаязыки. EBNF (Extended BNF) — один из них [52].

EBNF — расширение BNF, ориентированное на облегчение описания списков и необязательных элементов. EBNF не предоставляет дополнительных возможностей, поэтому все, что определяется в EBNF, может быть также определено в BNF.

Для представления списков важны следующие характеристики: нулевая длина, роль разделителя элементов. Разделитель отделяет элементы, если он появляется между ними. Разделитель завершает элементы, если он появляется после каждого элемента. Разделитель «завершения» более универсален.

В EBNF скобки `{ }` представляют нуль или более повторений заключаемой в них строки. Поэтому конструкция `{<оператор>;}` представляет нуль или более операторов, завершаемых точкой с запятой.

Правило подстановки EBNF

```
<список операторов> ::= {<оператор>; }
```

эквивалентно паре правил подстановки в BNF

```
<список операторов> ::= <empty>
                      | <оператор>; <список операторов>
```

Такие расширения BNF являются сокращениями, они могут уменьшить количество правил подстановки и нетерминалов, но они не изменяют то, что может быть определено.

Другим общепринятым расширением является использование скобок [и] для заключения в них необязательной конструкции. Например, необязательная целая часть в вещественном числе может быть описана так:

```
<веществ_число> ::= [<целая_часть>] . <дробн_часть>
```

Эквивалентом этого правила является пара BNF-правил:

```
<веществ_число> ::= <целая_часть> . <дробн_часть>
                  | . <дробн_часть>
```

Характерный состав метасимволов EBNF:

- фигурные скобки { и } обозначают нуль и более повторений;
- квадратные скобки [и] обозначают необязательную конструкцию;
- вертикальная черта | обозначает выбор;
- скобки (и) обозначают группировку.

EBNF имеет больше метасимволов, чем BNF. Кроме того, эти же символы могут также появляться в синтаксисе языка — индекс `i` в элементе `A[i]` вовсе не является необязательным — таким образом, требуется осторожность в различении обычных символов и метасимволов. Путаницу можно устранить заключением лексем в одиночные апострофы '('.

Пример. Грамматику арифметических выражений в BNF:

```
<выражение>  ::=  <выражение> + <терм>
                | <выражение> - <терм>
                | <терм>
<терм>  ::=  <терм> * <фактор>
                | <терм> / <фактор>
                | <фактор>
<фактор>  ::=  number
                | name
                | (<выражение>)
```

можно переписать в EBNF:

```
<выражение>  ::=  <терм> {(+ | -) <терм>}
<терм>  ::=  <фактор> {(* | / ) <фактор>}
<фактор>  ::= '('<выражение>')' | name | number
```

ПОЯСНЕНИЯ к примеру грамматики в EBNF

1. Здесь `<терм> {(+ | -) <терм>}` представляет последовательность одного или более термов, отделяемых символами + или -. Круглые скобки вокруг + | - гарантируют, что вертикальная черта представляет выбор между + и -.
2. В правой части правила для фактора скобки взяты в апострофы, так как это обычные символы.

В качестве дополнительного примера приведем правило EBNF для условного оператора языка Ada, который допускает многократное вложение в `then`-часть ветвей с условиями, а также имеет необязательную `else`-часть:

```
<if_оператор> ::=
if <выр-е> then <SL> {elsif <выр-е> then <SL>}
                     [else <SL>] end if;
```

ПРИМЕЧАНИЕ

В фигурные скобки заключается часть, которая может повторяться нуль и более раз. В квадратные скобки заключается необязательная часть.

Как видим, это синтаксическое правило тоже имеет очень лаконичную форму.

Синтаксические схемы

Синтаксические схемы представляют графическую нотацию грамматик, визуальную форму их записи. Другие названия схем: синтаксический граф, синтаксическая диаграмма. Придуманы они в корпорации Burroughs (1961) для описания языка Алгол.

Синтаксическая схема состоит из подсхем, определенных для каждого нетерминала грамматики.

Каждый путь в подсхеме соответствует правилу подстановки для нетерминала. Иначе говоря, каждое правило представляется в виде некоторой траектории от расположенной слева *точки входа* в подсхему до расположенной справа *точки выхода* из подсхемы. Компонентами пути-траектории являются терминалы и нетерминалы из правой части правила подстановки.

Терминалы заключаются в закругленные прямоугольники, нетерминалы — в обычные прямоугольники [9, 92, 93].

Синтаксическая схема, соответствующая BNF-грамматике для арифметических выражений, изображена на рис. 6.19.

В этой схеме четко различимы три подсхемы: для выражения, терма и фактора. Вход в каждую подсхему подписан именем соответствующего нетерминала. В табл. 6.8 приведены строки, порожденные различными траекториями в подсхемах. Отметим, что пути в подсхемах для выражения и терма имеют циклы, а в подсхеме для фактора цикла нет, поскольку он здесь не нужен.

Таблица 6.8. Применение подсхем

Название подсхемы	Примеры применения подсхем
выражение	терм терм + терм терм + терм + терм
терм	фактор фактор * фактор фактор * фактор / фактор
фактор	(выражение) name number

Преимущество синтаксических схем — в них используют только значащие нетерминалы. В BNF для создания альтернативных путей и циклов иногда требуются вспомогательные нетерминалы.

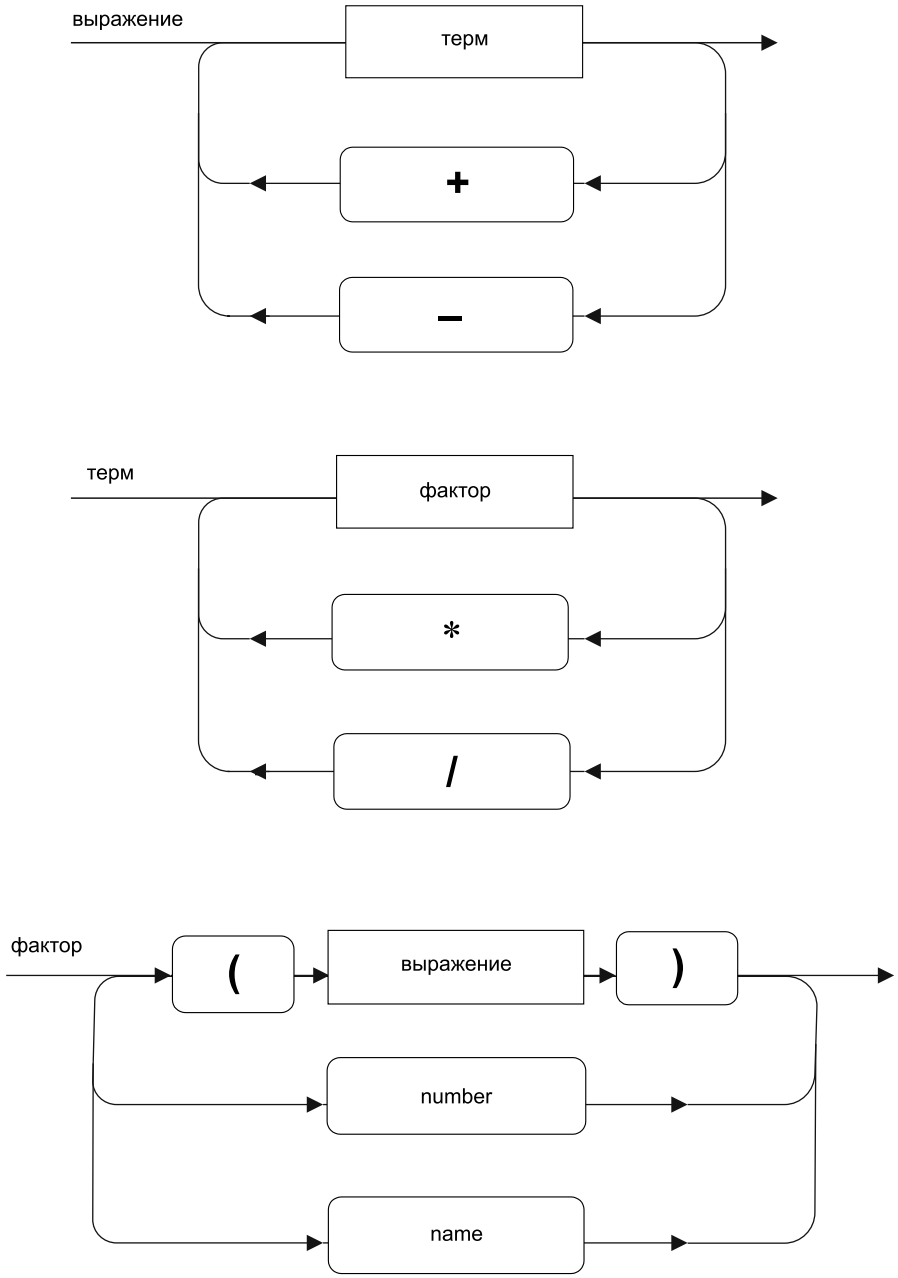


Рис. 6.19. Синтаксическая схема грамматики арифметических выражений

Точки с запятой и пустые операторы

В качестве назидательного примера рассмотрим влияние «синтаксических мелочей» на правильность использования языка.

Обратимся к фрагменту грамматики для операторов языка Pascal:

```
<S> ::= <empty>|оп-р|begin <SL> end
        |if <выр-е> then <S>
        |if <выр-е> then <S> else <S>
        |while <выр-е> do <S>
<SL> ::= <SL>; <S>|<S>
```

Как видим, в этом языке точки с запятой используют для отделения операторов друг от друга, то есть можно записать такой составной оператор:

```
begin оп-р1; оп-р2; оп-р3 end
```

Дерево разбора строки `begin оп-р1; оп-р2; оп-р3 end` представлено на рис. 6.20.

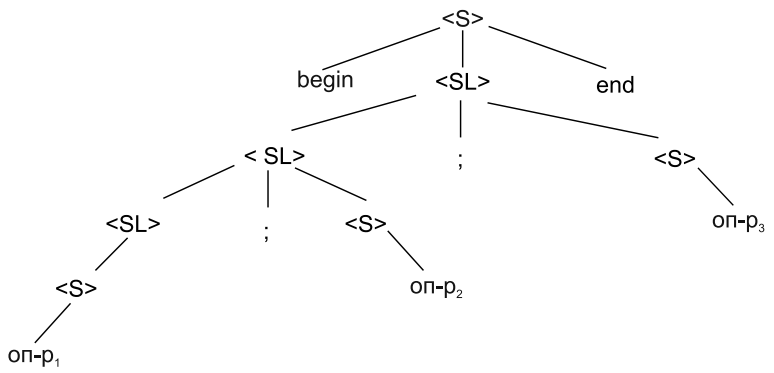


Рис. 6.20. Дерево разбора строки begin оп-р1; оп-р2; оп-р3 end

На первый взгляд, правило подстановки для пустого оператора позволяет точке с запятой превратиться из разделителя в терминатор, так как из строки

```
begin <S1>; <S2>; <S3>; <S4> end
```

можно легко получить строку

```
begin <S1>; <S2>; <S3>;          end
```

где принято, что `<S4>` генерирует пустую строку.

Следствием применения правила `<S> ::= <empty>` является исчезновение оператора `<S>`. Это исчезновение разрешает вставить точку с запятой как после, так и перед оператором:

```
begin ; оп-р1 ; ; оп-р2 ; ; ;  end
```

Однако такая вставка может исказить смысл Pascal-программы.

Например, фактически для условного оператора вставка точки с запятой после ключевого слова `then` подобна смерти. Ведь она эквивалентна помещению в `then`-ветвь пустого оператора!

Ну, а с точки зрения синтаксиса все в порядке и строка

```
if <вып-е> then; <operator>
```

корректна и эквивалентна двум строкам:

```
if < вып-е > then begin  end;
<operator>
```

Вывод простой: мелочей в синтаксисе нет, любая недоработка может откликнуться серьезными проблемами написания и выполнения программ.

Контрольные вопросы и упражнения

1. Составить абстрактные синтаксические деревья для следующих выражений:

```
а) a * b + c
б) a * (b + c)
в) a * b + c * d
г) a * (b + c) * d
д) [b / 2 + sqrt((b / 2) * (b / 2) – a * c)] / a
```

2. Получить контекстно-свободные грамматики, описывающие синтаксис следующих конструкций:

   а) строки (с длиной один или более), составленные из набора символов-терминалов `break`, `tab`, `newline`;

   б) последовательности букв или цифр, начинающиеся с буквы;

   в) вещественные числа, в которых целая или дробная часть может быть пустой (но не обе части). Грамматика должна разрешать запись вида 31., 3.1, .13, но не просто десятичную точку.

3. Дана BNF-грамматика для вещественных чисел:

```
<вещ_число> ::= <цел_ч>.<дроб_ч>
<цел_ч> ::=  <цифра> | <цел_ч> <цифра>
<дроб_ч> ::= <цифра> | <цифра> <дроб_ч>
<цифра> ::=  0| 1| 2| 3| 4| 5| 6| 7| 8| 9
```

   Используя данную грамматику, необходимо получить левосторонний вывод для строк 5.176, 18.384.

4. Используя грамматику для арифметических выражений:

```
<E> ::=  <E> + <T> | <E> – <T> | <T>
<T> ::=  <T> * <F> | <T> / <F> | <F>
<F> ::=  number | name | (<E>)
```

   где `<E>` — выражение, `<T>` — терм, `<F>` — фактор, изобразить дерево разбора для следующих арифметических выражений:

```
а) 2 + 3
б) (2 + 3)
в) 2 + 3 * 5
г) (2 + 3) * 5
д) 2 + (3 * 5)
```

5. Изобразить абстрактные синтаксические деревья для выражений из 4-го упражнения. Сравнить их с соответствующими деревьями разбора.

6. Следующие BNF-правила описывают конструкции, которые состоят из списка каких-то элементов. Необходимо описать список и дать ответы на следующие вопросы:

   - как много элементов он может иметь?
   - появится ли что-нибудь до, между или после элементов?

```
а) <список_имен> ::= <имя> | <имя> , <список_имен>
б) <список_полей> ::= <поле> ; | <поле> ; <список_полей>
в) <список_операторов> ::= <empty> | <оператор> ; <список_операторов>
г) <терм> ::=  <терм> * <фактор> | <фактор>
д) <переменные> ::= <empty> | var <объявление_переменной>
     <объявление_ переменной> ::= <список_имен> : <тип> ;
      | <список_имен> : <тип> ; <объявление_ переменной>
е) <константы> ::= <empty> | <объявление_константанты>
     <объявление_константы> ::= const  <имя> = <константа> ;
      | <объявление_константы> <имя> = <константа>;
```

7. Таблица операций языка Pascal имеет следующий вид.

Операции	Пояснения	Нетерминал
<, <=, =, < >, >=, >, in	Операции отношений и проверки принадлежности	<выражение_отношения>
+ , – , or	Аддитивные операции	<выражение>
*, /, div, mod, and	Мультипликативные операции	<терм>
not	Логическое отрицание	<фактор>

   Все операции, отображенные в таблице, имеют левую ассоциативность (группировка элементов осуществляется слева направо). Все операции в одной строке имеют одинаковый приоритет.

   Записать грамматику выражений для языка Pascal.
   Расширить полученную грамматику, добавив одноместную операцию изменения знака выражения (например: `-1`, `-(a - b)`, но не `2 + -3`):

```
< выражение > :: = - < терм >
```

8. Используя грамматику, полученную в седьмом упражнении, создать дерево разбора для следующих выражений:

```
а) i >= 0
б) (i >= 0) and not p
в) i >= a and not p
г) (i >= 0) and (x <> y)
```

9. Записать грамматику выражений для C, используя таблицу операций этого языка.

10. Используя грамматику из упражнения 9, создать дерево разбора для следующих выражений:

```
а) i >= 0
б) (i>= 0) && ! p
в) i >= 0 && ! p
г) i >= 0 && x != y
```

11. Грамматика, основанная на синтаксисе операторов языка Pascal, имеет вид:

```
<S>::= <идентификатор>:= <выражение>
    | if <выражение>then <S>
    | if <выражение>then <S> else <S>
    | while <выражение> do <S>
    | begin <SL> end
<SL>::= <S> | <S> ; <SL>
```

где `<SL>` — список операторов.

Создать дерево разбора для каждой из конструкций

```
while выражение do идентификатор:= выражение
begin идентификатор:= выражение end
if выражение then if выражение then S else S
```

12. Даны BNF-правила для заголовка процедуры и функции языка Pascal. Построить EBNF-правила и синтаксические схемы.

BNF-правила:

```
<заг_проц> ::= procedure <идент>
    | procedure <идент> (<список_парам>)
<список_парам> ::= <форм_парам> | <форм_парам>; <список_парам>
<форм_парам>::= <парам_проц>|<парам_функц>
<парам_проц>::= <парам_перем>|<парам_знач>
<парам_функц>::= <парам_знач>
<парам_перем>::= var <список_идент>:<идент_типа>
<парам_знач>::= <список_идент>:<идент_типа>
<список_идент>::= < идент > | < идент >, <список_идент>
<заг_функц>::= function < идент> : < идент_типа >
    | function < идент> (<список_парам>) : < идент_типа >
```

Решение

EBNF-правила:

```
<заг_проц> ::= procedure <идент> ['('<форм_парам>{; <форм_парам>}')']
<форм_парам>::=  <парам_проц>|<парам_функц>
<парам_проц>::= <парам_перем>|<парам_знач>
<парам_функц>::= <парам_знач>
<парам_перем>::= var <список_идент>:<идент_типа>
<парам_знач>::= <список_идент>:<идент_типа>
<список_идент>::= < идент >{,< идент >}
<заг_функц>::= function < идент> ['('<форм_парам >
    {;<форм_парам >}')'] : < идент_типа >
```

13. Дана грамматика:

```
<опер_присв>::= <идент>:= <выр-е>
<выр-е>::= <идент> + <выр-е>| <идент> * <выр-е> | (<выр-е>)|<идент-р>
<идент>::= A|B|C
```

а) Необходимо получить левосторонний вывод (заменой самого левого нетерминала) для строки `A:=A * (B+(C * A))`.

б) Изобразить дерево, соответствующее полученному выводу.

14. Дана грамматика:

```
<опер-р_присв>::= <идент>:= <выр-е>
<выр-е>::= <выр-е> + <терм >| <терм >
<терм >::= <терм > * <фактор>|<фактор>
```

```
<фактор>::= (<выр-е>)|<идент>
<идент>::= A|B|C
```

а) Необходимо получить левосторонний вывод (заменой самого левого нетерминала) для строки `A:= (A + B) * C`.

б) Нарисовать дерево, соответствующее полученному выводу.

15. Доказать, что следующая грамматика является неопределенной (неоднозначной):

```
<S>::= <A>
<A>::= <A> + <A> | < идент >
< идент >::= a | b | c
```

16. Описать язык, определенный грамматикой:

```
<S>::= <A> <B> <C>
<A>::= a <A> | a
<B>::= b <B> | b
<C>::= c <C> | c
```

Ответ: одна или более «a», за которой следует одна или более «b», за которой следует одна или более «c».

Используя данную грамматику, построить деревья разбора для предложений aabcc, aaabbbc.

17. Написать грамматику для языка из строк, которые имеют «n» копий буквы «a», за которой следует то же количество копий буквы «b», где $n > 0$. Например, допустимы строки: aaaabbbb, aaaaaaabbbbbbb, но не допустимы строки вида a, abb, ba, aaab. (Для проверки полученной грамматики строим дерево.)

```
<S>::= a <S> b | a b
```

Глава 7

Формальная семантика языков программирования

Эта глава отведена обсуждению формальных подходов к описанию семантики языков программирования. Вначале поясняются синтезируемые атрибуты и атрибутные грамматики, которые являются расширениями обычных, синтаксических грамматик на область семантики. Затем дается характеристика операционной семантики, максимально задействующей воображение человека в роли виртуальной машины. Значительное место и внимание уделено аксиоматической семантике — наиболее развиваемому на практике подходу к автоматическому доказательству корректности программы. Рассматривается принятая в нем схема доказательства, использование правил вывода, опирающихся на постусловия и предусловия; приводятся примеры подобных правил для наиболее популярных конструкций языка. В частности, описывается понятие инварианта цикла и исследуются методики его определения. В заключение объясняются основные идеи денотационного подхода к заданию семантики, где во главу угла поставлены рекурсивные функции отображения отдельных конструкций языка в область семантических значений. Демонстрируются примеры функций семантического отображения для чисел, выражений, операторов присваивания и циклов.

Семантика языка программирования

Считается, что в настоящее время средства для определения синтаксиса языка достаточно удобны и развиты. Увы, но этого нельзя сказать о средствах формального представления семантики языка. Понятно, что для использования языка программирования следует описать каждую конструкцию языка в отдельности, а также ее применение в совокупности с другими конструкциями. В языке существует множество различных конструкций, точное определение которых необходимо как программисту, применяющему язык, так и разработчику компилятора для этого языка. Программисту эти знания позволяют прогнозировать вычисления, производимые операторами программы. Разработчику описания конструкций необходимы для создания правильной реализации компилятора.

Обычно семантика определяется с помощью обычного текста. Сначала при помощи какой-либо грамматики (скажем, BNF-грамматики) описывается синтак-

сис конструкции, а затем приводятся примеры и комментарии для разъяснения семантики. Очень часто смысл комментариев неоднозначен, поэтому различные читатели воспринимают его по-разному. Программист может получить ошибочное представление о работе создаваемой программы, а разработчик может сформировать ошибочную реализацию некоторой конструкции языка. Для исключения ошибок нужен инструмент для ясного, точного и лаконичного определения семантики всего языка.

Приемлемое решение в отношении семантики языка найти чрезвычайно трудно. Причина кроется в том, что содержание, смысл конструкций во много раз сложнее их формы. Был разработан целый ряд методов для формального определения семантики. Приведем описания наиболее популярных из них.

Синтезируемые атрибуты

Ранние попытки точно задать семантику делались путем добавления расширений к BNF-грамматике языка программирования. Дополнительную информацию о семантике можно было извлечь из дерева грамматического разбора. Идея, предложенная Дональдом Кнутом, заключалась в том, чтобы сопоставить каждому узлу дерева разбора программы некоторую функцию, определяющую семантику этого узла [70]. Подобная информация сохранялась в так называемых атрибутах.

Семантика конструкции может представляться некоторой величиной или набором величин, связанных с конструкцией. Например, семантика выражения `3 + 4` может быть целым значением `7`, типом `int` или строкой `+ 3 4`.

Величина, ассоциированная с конструкцией, называется *атрибутом*. Атрибут *a* для *X* будем записывать как *X.a*, где *X* считается нетерминалом или терминалом грамматики. *E.val* рассматривается как ссылка на атрибут *val* выражения *E*. На рис. 7.1 все символы грамматики имеют по атрибуту *val*. В общем случае каждый символ грамматики может иметь несколько атрибутов.

Атрибуты имеют значения. Точнее, каждое вхождение атрибута в дерево разбора имеет значение. Атрибут в корне дерева разбора на рис. 7.1 имеет значение `11`. Атрибут **num**.*val* слева внизу рисунка имеет значение `7`.

ПРИМЕЧАНИЕ

> В этой главе будем использовать упрощенное обозначение нетерминалов и терминалов грамматики. Нетерминалы будем записывать курсивом (без угловых скобок), а терминалы — полужирным шрифтом.

Будем полагать, что атрибуты для терминальных символов поступают вместе с символами. Когда лексический анализатор распознает на входе знак **num**, он также определяет значение знака — предполагаем, что это значение «приходит» вместе с **num** в атрибуте **num**.*val*.

Значения атрибутов для нетерминалов определяются семантическими правилами, добавленными к правилам подстановки грамматики. В этом разделе мы будем использовать синтезируемые атрибуты. Атрибут *N.a* называется *синтези-*

руемым, если правила, определяющие *N.a*, добавляются к правилам подстановки, в которых *N* появляется в левой части правила. Иными словами, синтезируемый атрибут в узле *N* дерева разбора определяется только с использованием значений атрибутов в дочерних по отношению к *N* узлах и в самом узле *N*. Семантическое правило, определяющее *N.a*, записывается как присваивание атрибуту *N.a* некоторого значения.

Использование синтезируемых атрибутов известно как синтаксически управляемый подход к определению семантики. Спецификация синтаксиса совместно со связанными с ней семантическими правилами называется *синтаксически управляемым определением*.

Синтаксически управляемое определение в табл. 7.1 основывается на грамматике для выражений. Каждый символ грамматики имеет синтезируемый атрибут *val*. С каждым правилом подстановки связано одно семантическое правило (это правило записано в той же строке). Здесь T и T_1 обозначают ссылки на два вхождения одного и того же нетерминального символа в правиле.

Аннотированное дерево разбора имеет атрибуты, добавленные к узлам. Аннотированное дерево разбора на рис. 7.1 создано применением правил подстановки и семантических правил из табл. 7.1.

Таблица 7.1. Синтаксически управляемое определение для вычисления арифметических выражений

Правило подстановки	Семантическое правило
$E ::= E_1 + T$	$E.val := E_1.val + T.val$
$E ::= E_1 - T$	$E.val := E_1.val - T.val$
$E ::= T$	$E.val := T.val$
$T ::= T_1 * F$	$T.val := T_1.val * F.val$
$T ::= F$	$T.val := F.val$
$F ::= (E)$	$F.val := E.val$
$F ::= \mathbf{num}$	$F.val := \mathbf{num}.val$

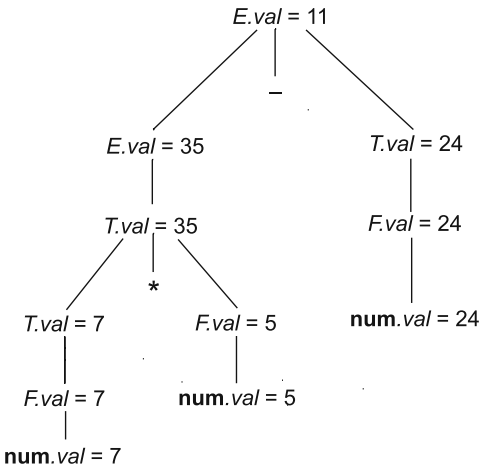


Рис. 7.1. Аннотированное дерево разбора с атрибутами

Листья дерева разбора помечены терминалами. Терминалы тоже могут иметь атрибуты; правда, семантических правил для определения значений терминалов нет. При обходе левой ветви дерева разбора снизу вверх правило

```
F.val := num.val
```

определяет, что значение атрибута *val* для самого левого узла *F* равно 7. Правило

```
T.val := F.val
```

связанное с правилом подстановки $T ::= F$, копирует это значение, так что *T.val* тоже равно 7.

Более высокий, родительский узел *T* генерирует правило подстановки

```
T ::= T1 * F
```

поэтому его значение является произведением значений T_1 и *F*. Следовательно, левый дочерний узел корня имеет значение 35. Семантическое правило, связанное с правилом подстановки для корня, определяет, что значение *E* должно быть разностью значений ее детей.

Порядок вычислений

При использовании синтезируемых атрибутов информация передается вверх по дереву разбора. Значение атрибута узла определяется в терминах атрибутов ее дочерних узлов. Именно поэтому значения атрибутов на рис. 7.1 могут быть вычислены обходом дерева от листьев к корню.

Выводы

Предусматривается следующий порядок использования синтезируемых атрибутов для определения смысла конструкций:

1. *Атрибуты* — атрибуты добавляются к символам грамматики.
2. *Семантические правила* — семантические правила добавляются к правилам подстановки. Синтезируемый атрибут *N.a* определяется по семантическому правилу, добавленному к такому правилу подстановки, в левой части которого находится *N*.

Атрибутные грамматики

Атрибутные грамматики можно рассматривать как обобщение аппарата синтезируемых атрибутов [70]. Они позволяют осмысливать конструкции в зависимости от окружающего контекста. В этом случае значения атрибутов передаются как вверх, так и вниз по дереву разбора. Дело в том, что смысл узла на дереве разбора может зависеть не только от его поддерева, но и от информации из другой части дерева.

Предусматривается следующий порядок использования атрибутных грамматик для определения смысла конструкций:

1. *Атрибуты* — атрибуты добавляются к символам грамматики. Для каждого атрибута определяется, является ли он синтезируемым или наследуемым.

2. *Семантические правила* — семантические правила добавляются к правилам подстановки.
3. Если нетерминал *N* появляется в левой части правила подстановки, то добавляемые семантические правила определяют *синтезируемые* атрибуты для *N*.
4. Если нетерминал *A* появляется в правой части правила подстановки, то добавляемые семантические правила определяют *наследуемые* атрибуты для *A*. Именно наследуемые атрибуты инициируют передачу информации вниз по дереву разбора или по горизонтали этого дерева (в этом случае говорят о передаче информации от «братских» узлов).

Рассмотрим проблему трансляции десятичных чисел в диапазоне от 0 до 99 в русские фразы (табл. 7.2).

Таблица 7.2. Трансляция десятичных чисел

Число	Фраза
0	Ноль
1	Один
10	Десять
19	Девятнадцать
20	Двадцать
29	Двадцать девять
30	Тридцать
31	Тридцать один

Трансляция `тридцать один` для числа `31` строится на основе фразы `тридцать`, трансляции цифры `3` в левой позиции числа и трансляции цифры `1` в правой позиции числа. Однако есть исключения из этого правила. Трансляцией числа `30` является `тридцать`, а не `тридцать ноль`. Трансляцией числа `19` является фраза `девятнадцать`, а не `десять девять`.

Проиллюстрируем применение наследуемых атрибутов для трансляции чисел от 0 до 99, генерируемых по следующей грамматике:

```
N :: = D|D P
P ::= D
D :: = 0|1|2|3|4|5|6|7|8|9
```

Дерево разбора для числа 68 представлено на рис. 7.2, *а*.

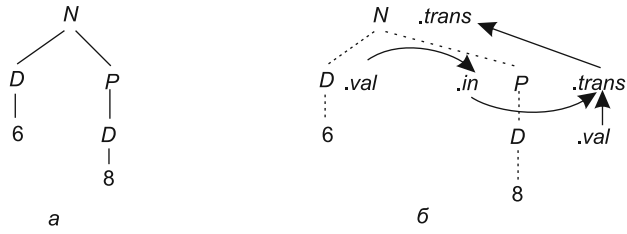


Рис. 7.2. Дерево разбора для числа 68

Нетерминал *D* имеет синтезируемый атрибут *val*, который представляет значение цифры, генерируемой по правилу подстановки для *D*. Нетерминал *N* имеет синте-

зируемый атрибут *trans*, который дает трансляцию числа, генерируемую по правилу для *N*. Нетерминал *P* имеет наследуемый атрибут *in* и синтезируемый атрибут *trans*.

Стрелки на рис. 7.2, *б* иллюстрируют, как синтезируемая для узла *P* трансляция представляется в виде атрибута *P.trans*. Для создания *P.trans* используются значения обеих цифр. С правилом подстановки *P* ::= *D* ассоциируется семантическое правило, которое определяет *P.trans* в терминах *P.in* (значения левой цифры) и *D.val*. На рис. 7.2, *б P.in* = 6 и *D.val* = 8, а *P.trans* является фразой для 68 = 10 * *P.in* + *D.val*.

Полная атрибутная грамматика представлена в табл. 7.3. С правилом подстановки *N* :: = *D P* связаны два семантических правила:

```
P.in := D.val
N.trans := P.trans
```

Словом, *P.in* наследует значение *D.val* от левой цифры, а *P.trans* становится трансляцией *N.trans* (почти волшебным образом превращается в эту трансляцию).

Таблица 7.3. Атрибутная грамматика

Правило подстановки	Семантические правила
`N ::= D`	`N.trans := spell(D.val)`
`N ::= D P`	`P.in := D.val` `N.trans := P.trans`

`P ::= D`	`P.trans := if D.val = 0 then decade(P.in)`
	`else if P.in ≤ 1 then spell(10*P.in + D.val)`
	`else decade(P.in) • spell(D.val)`
	здесь точка • обозначает операцию конкатенации (составления строки из отдельных частей)

`D ::= 0`	`D.val:= 0`
`D ::= 1`	`D.val:= 1`
…	…
`D ::= 9`	`D.val:= 9`

С правилом подстановки *P* :: = *D* связано семантическое правило, которое определяет *P.trans*. Это правило основывается на следующем псевдокоде для трансляции числа *n*:

```
if n is a multiple of 10 then decade(n div 10)
     else if n < 20 then spell(n)
          else decade(n div 10) • spell(n mod 10)
```

Функции *spell* и *decade* удовлетворяют следующим уравнениям:

```
spell(1) = один, spell(2) = два, …, spell(19) = девятнадцать
decade(0) = ноль, decade(1) = десять, …, decade(9) = девяносто
```

С правилом подстановки *P* :: = *D* связано следующее правило:

```
P.trans := if D.val = 0 then decade(P.in)
           else if P.in ≤ 1 then spell(10*P.in + D.val)
                else decade(P.in) • spell(D.val)
```

Синтезируемый атрибут *P.trans* определяется в терминах наследуемого атрибута *P.in* и синтезируемого атрибута *D.val*.

Наследуемые атрибуты придают потоку информации ясность и недвусмысленность. Впрочем, без них можно обойтись, поскольку все, что определяется с помощью

наследуемых и синтезируемых атрибутов, можно определить с использованием лишь синтезируемых атрибутов. Например, атрибутная грамматика из табл. 7.3 может быть модифицирована в эквивалентную грамматику, применяющую только синтезируемые атрибуты. Мы ввели эту грамматику лишь затем, чтобы проиллюстрировать, как можно использовать наследуемые атрибуты.

Операционная семантика

Операционное определение языка программирования описывает выполнение программы, составленной на данном языке, средствами виртуального компьютера [86]. Виртуальный компьютер определяется как абстрактный автомат. Внутренние состояния этого автомата моделируют состояния вычислительного процесса при выполнении программы. Автомат транслирует исходный текст программы в набор формально определенных операций. Этот набор задает переходы автомата из исходного состояния в последовательность промежуточных состояний, изменяя значения переменных программы. Автомат завершает свою работу, переходя в некоторое конечное состояние. Таким образом, здесь идет речь о достаточно прямой абстракции возможного использования языка программирования.

Итак, операционная семантика описывает смысл программы путем выполнения ее операторов на простой машине-автомате. Изменения, происходящие в состоянии машины при выполнении данного оператора, определяют смысл этого оператора.

С одной стороны, машина-автомат способна воспринимать задания, представляемые лишь на простом языке моделирования. С другой стороны, современный компьютер может служить универсальным интерпретатором такого языка. Подобный интерпретатор может быть реализован программой, которая становится виртуальной машиной для языка моделирования.

Таким образом, в операционной семантике требуются:

- транслятор, преобразующий операторы исходного языка программирования **L** в команды низкоуровневого языка моделирования;
- виртуальная машина (ВМ) для языка моделирования.

Например, язык моделирования может представляться набором простых команд, записанных в табл. 7.4.

Таблица 7.4. Язык моделирования виртуальной машины

Команды виртуальной машины
`ident := var`
`ident := ident + 1`
`ident := ident – 1`
`goto  label`
`if var relop var goto label`
`ident := var bin_op var`
`ident := un_op var`

Здесь приняты следующие обозначения: `var` — переменная, `ident` — имя переменной, `goto label` — команда безусловного перехода на символический адрес `label`,

`relop` — обозначение одной из операций сравнения (<, >, =, <>), `bin_op` — операция над двумя аргументами, `un_op` — операция над одним аргументом. Условимся, что символический адрес команды (метку) будем отделять от команды двоеточием.

Положим, что нам нужно выяснить смысл оператора `for` для языка программирования C. Мы представляем этот оператор с помощью некоторой программы для виртуальной машины, приведенной в табл. 7.5.

Таблица 7.5. Программа для виртуальной машины

Оператор языка C	Команды виртуальной машины
`for (expr1; expr2; expr3)`	`expr1;`
`{`	`loop: if expr2 = 0 goto out`
`...`	`...`
`}`	`expr3;`
	`goto loop;`
	`out: …`

После этого запускаем нашу виртуальную машину. Смысл оператора определяют изменения в состоянии виртуальной машины, происходящие при выполнении команд — результата трансляции `for`-оператора. Подобные программы могут составляться и для последовательности операторов. Главное, чтобы имеющийся транслятор был способен обрабатывать каждый из используемых операторов.

ПРИМЕЧАНИЕ

Человек, читающий программы для виртуальной машины, тоже может считать себя виртуальным компьютером, то есть полагают, что он может правильно выполнять «команды» и распознает эффекты «выполнения».

В 1972 году была разработана операционная модель языка под названием Vienna Definition Language (VDL) — это метаязык, предназначенный для описания других языков. В ее состав входили транслятор и виртуальная машина. С помощью данной модели был описан язык программирования PL/1.

Аксиоматическая семантика

Данный подход основывается на применении аппарата исчисления предикатов и теории доказательств. Семантику каждой конструкции языка определяют как некий набор аксиом или правил вывода, используемый для вывода результатов выполнения этой конструкции. Чтобы понять смысл всей программы (то есть разобраться, что и как она делает), эти аксиомы и правила вывода следует применять так же, как при доказательстве обычных математических теорем. Аксиомы и правила вывода используют для оценки значений переменных после выполнения каждого оператора программы. В итоге, когда программа выполнена, формируется доказательство того, что вычисленные результаты удовлетворяют заданным ограничениям на их значения относительно входных значений. Словом, доказывается,

что выходные данные являются корректными результатами вычисления функций, обрабатывающих соответствующие входные данные. Примером описанного подхода считается метод *аксиоматической семантики*, созданный Тони Хоаром [66–68].

Итак, метод аксиоматической семантики служит для доказательства правильности программы. Доказательство правильности демонстрирует, что программа выполняет вычисления, описываемые ее спецификацией. При доказательстве каждый оператор окружается логическими выражениями (условиями), которые задают ограничения на программные переменные. Они используются для определения смысла оператора. Программа представляется в виде *хоаровских* троек (рис. 7.3).

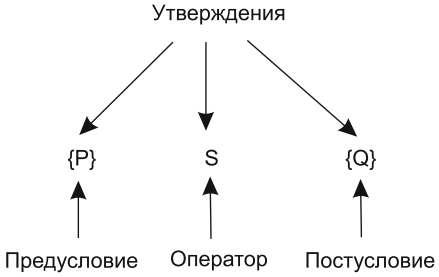


Рис. 7.3. Хоаровская тройка программы

Для получения тройки к каждому оператору нужно добавить слева предусловие, а справа — постусловие. Условия (утверждения, высказывания) записываются в фигурных скобках.

Предусловие описывает ограничения на программные переменные перед выполнением оператора и обозначается как `{P}`.

Постусловие задает новые ограничения на эти переменные после выполнения оператора и обозначается как `{Q}`.

Принято: предусловие вычисляется на основе постусловий операторов.

Среди множества предусловий особую роль играет слабейшее предусловие.

Слабейшее предусловие — это наименьшее ограничивающее предусловие, которое гарантирует правильность связанного постусловия.

Рассмотрим пример хоаровской тройки:

```
{x > 10}   result := 5 * x + 1   {result > 1}
{x > 50}
{x > 1000}
```

В примере записано, что при правильной работе оператора `result := 5 * x + 1` на результат накладывается ограничение `result > 1`.

Здесь перечислены три возможных предусловия, налагающих разные ограничения на значения «входной» переменной `x`, но слабейшее предусловие имеет следующий вид:

```
{x > 0}  --  слабейшее предусловие
```

Слабейшее предусловие все еще обеспечивает корректную работу оператора, при котором соблюдается постусловие, но при этом предоставляет наибольшую «свободу» переменной из предусловия. Ослабление предусловия можно проиллюстрировать рис. 7.4.

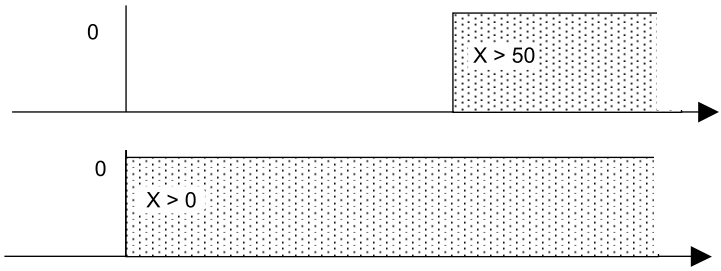


Рис. 7.4. Ослабление предусловия

Здесь показана числовая ось, на которой штриховкой обозначены области разрешенных значений для переменной x. Видим, что слабейшему предусловию соответствует максимальная область допустимых значений.

В аксиоматической семантике принята следующая схема доказательства программы:

1. Слабейшее предусловие оператора вычисляется на базе его постусловия.
2. Выполняется обратный проход по программе — доказательство начинается с последнего оператора, завершается первым оператором.
3. Первое предусловие задает условия, при которых программа формирует последнее постусловие (желаемые результаты).

Для некоторых операторов вычисление слабейшего предусловия выполняется просто и может быть определено аксиомой. В большинстве случаев слабейшее предусловие нужно вычислять по правилу вывода.

Аксиома — это логическое утверждение, принимаемое без доказательства в силу непосредственной убедительности.

Правило вывода — это метод получения истинного утверждения на основе других утверждений.

Аксиоматическая семантика требует, чтобы для каждого оператора языка программирования была определена или аксиома, или правило вывода.

Аксиома присваивания

Эта аксиома «обслуживает», то есть позволяет «доказать» оператор присваивания.

Пусть `x := E` — оператор присваивания с постусловием `Q`. Тогда его предусловие, определяемое по аксиоме

$P = Q_{X \to E}$

означает, что P вычисляется как Q, в котором все вхождения переменной x замещаются на E, то есть на выражение из правой части оператора.

Аксиома записывается в виде следующей тройки:

$\{P = Q_{X \to E}\}$ `x := E` $\{Q\}$

Повторим, что аксиоматическая семантика разрабатывается для доказательства правильности программ. С этой точки зрения оператор присваивания с предусло-

вием и постусловием может рассматриваться как теорема. Если аксиома присваивания при применении к постусловию и оператору вырабатывает «правильное» предусловие, то теорема доказана.

Применение аксиомы присваивания

Рассмотрим примеры применения аксиомы присваивания, то есть примеры доказательства теорем по поводу операторов присваивания.

Пусть исходные данные для доказательства теоремы имеют вид:

```
a := b/2 + 1 {a < 50}
```

Тогда для заданного оператора `a := b/2 + 1` слабейшее предусловие вычисляется подстановкой в постусловие `{a < 50}` вместо `a` выражения `b/2 + 1`:

```
{b/2 + 1 < 50} ⇒ {b < 98}
```

Как использовать этот результат? Достаточно проверить соблюдение предусловия и постусловия в ходе реальных вычислений.

Еще один пример. Для заданной пары

```
y := 5 * x – 10 {y > 95}
```

слабейшее предусловие равно:

```
{5 * x – 10 > 95} ⇒ {x > 17}
```

Правило консеквенции (упрощения)

Рассмотрим хоаровскую тройку `{a > 7} a := a – 2 {a > 0}`. В этом случае утверждение, вырабатываемое по аксиоме присваивания, не равно предусловию. И все же очевидно, что из `{a > 7}` следует `{a > 2}`. Для использования этого факта в доказательстве нам потребуется правило вывода «*правило консеквенции*», иначе называемое *правилом упрощения*.

Общая форма правила вывода имеет вид

$$\frac{S1, S2, ..., SN}{S},$$

где утверждается, что если исходные высказывания $S1$, $S2$, ..., SN истинны, то может быть выведена истина для заключения S.

Правило *консеквенции* записывается в форме

$$\frac{\{P\}S\{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q'\}},$$

где символ $\Rightarrow$ обозначает «следует», а S — любой оператор программы.

Правило означает: если логическое высказывание $\{P\}\ S\ \{Q\}$ — истина, а из утверждения P' следует утверждение P и из утверждения Q следует утверждение Q', то может быть выведено истинное логическое высказывание $\{P'\}\ S\ \{Q'\}$.

Это правило говорит, что *постусловие* может быть всегда *ослаблено*, а *предусловие* может быть всегда *усилено*.

Такое ослабление/усиление очень полезно в доказательстве программ. Например, это позволяет завершить доказательство начального логического высказывания.

Примем P = `{a > 2}`, P' = `{a > 7}`, $Q = Q'$ = `{a > 0}`,
тогда

$$\frac{\{a > 2\}\ \ a := a - 2\ \ \{a > 0\}, \{a > 7\} \Rightarrow \{a > 2\}, \{a > 0\} \Rightarrow \{a > 0\}}{\{a > 7\}\ \ a := a - 2\ \ \{a > 0\}}.$$

Первое исходное утверждение `{a > 2} a := a – 2 {a > 0}` доказывается по аксиоме присваивания. Второе и третье утверждения очевидны. Следовательно, по правилу консеквенции, такое упрощение является истинным.

Правило вывода для последовательности

Слабейшее предусловие для последовательности операторов не может описываться аксиомой, так как предусловие зависит от конкретных разновидностей операторов в последовательности. В этом случае предусловие может быть описано только с помощью правила вывода.

Пусть $S1$ и $S2$ — два смежных оператора, имеющих следующие пред- и постусловия: $\{P1\}\ S1\ \{P2\}$, $\{P2\}\ S2\ \{P3\}$. Отметим, что постусловие первого оператора совпадает с предусловием второго. Логически это вполне объяснимо: первый оператор формирует данные для успешного старта второго оператора. Тогда правило вывода для последовательности из двух операторов принимает вид:

$$\frac{\{P1\}\ \ S1\ \ \{P2\}, \{P2\}\ \ S2\ \ \{P3\}}{\{P1\}\ \ S1; S2\ \ \{P3\}}.$$

Для данного примера $\{P1\}\ S1; S2\ \{P3\}$ правило вывода предписывает аксиоматическую проверку последовательности операторов $S1$; $S2$: они должны иметь общее утверждение, иначе ограничение на переменные, — это $\{P2\}$. Мало того, предусловие первого оператора должно стать общим предусловием последовательности, а постусловие второго — общим постусловием.

Правило вывода для последовательности имеет очень большое практическое значение. Именно оно обеспечивает цепочку автоматического доказательства операторов программы. Для запуска цепочки достаточно знать лишь требуемый результат программы, то есть постусловие последнего оператора. При доказательстве последнего оператора будет вычислено его слабейшее предусловие. Это даст возможность перейти к доказательству предпоследнего оператора. Ведь правило вывода для последовательности гласит: предусловие последнего оператора равно постусловию предпоследнего. Далее шаги перехода от оператора к оператору повторяются. Процесс завершается доказательством первого оператора программы, то есть вычислением его слабейшего предусловия, которое определяет ограничения на исходные данные программы.

Применение правила вывода для последовательности

Пусть дана программа из двух операторов, причем для нее записано постусловие:

```
a := 2 * c - 1;
b := a + 5;
Q = {b < 10}
```

По аксиоме присваивания предусловие для 2-го оператора `b := a + 5` равно `{a < 5}`.

Правило для последовательности позволяет использовать это предусловие как постусловие для 1-го оператора `a := 2 * c - 1`.

Тогда соответствующее предусловие 1-го оператора и всей программы (по аксиоме присваивания) равно:

```
P = {2 * c - 1 < 5} ⇒ {c < 3}
```

Правило вывода для условного оператора

Данное правило позволяет получить семантически правильный условный оператор с булевым условием ветвления B, оператором $S1$, который надо поместить в `then`-ветвь, и оператором $S2$, который следует разместить в `else`-ветви.

У правильного условного оператора должна быть одна точка входа и одна точка выхода. Это означает, что постусловие у $S1$ и $S2$ должно быть одинаковое, например Q.

Предусловия операторов $S1$ и $S2$ описываются достаточно сложно. С одной стороны, у них есть общая часть P, которая задает стартовые ограничения на «арифметические» переменные. С другой стороны, оператор $S1$ должен запускаться при истинном значении булева условия B, а оператор $S2$ — при его ложном значении. Отсюда вывод: предусловия должны быть представлены как логические перемножения прямого (инверсного) булева условия B (not B) и утверждения P:

$$\frac{\{B \ and \ P\} \ S1 \ \{Q\}, \{(not \ B) \ and \ P\} \ S2 \ \{Q\}}{\{P\} \ if \ B \ then \ S1 \ else \ S2 \ \{Q\}}.$$

Как видим, аксиоматическая семантика условного оператора не оставляет без внимания ни одной детали, связанной с его компонентами.

Применение правила вывода для условного оператора

Положим, что стоит задача проверки возможности построения следующего условного оператора:

```
if (a > b) then m := m + 3 else m := m - 3;
Q = {m > 0}
```

Булевым условием ветвления здесь является выражение отношения `a > b`.

Кандидатом в `then`-часть служит оператор `m := m + 3`, а кандидатом в `else`-часть — оператор `m := m – 3`.

Учитывая, что постусловие `{m > 0}` у них общее, определим слабейшие «арифметические» предусловия:

- Для `then`-оператора `{m + 3 > 0}` = `{m > –3}`.
- Для `else`-оператора `{m – 3 > 0}` = `{m > 3}`.

Видим, что несовпадение налицо. Но оно преодолимо. Поскольку справедливо `{m > 3}` ⇒ `{m > –3}`, то по правилу консеквенции можно использовать `{m > 3}` как предусловие всего условного оператора.

Таким образом, из рассмотренных компонентов можно собрать корректный условный оператор.

Вычисление предусловия для цикла FOR

Количество итераций цикла `FOR` заранее известно, поэтому данный цикл может трактоваться как повторение последовательности операторов его тела.

Вычисление предусловия здесь может выполняться обратной разверткой цикла (рис. 7.5).

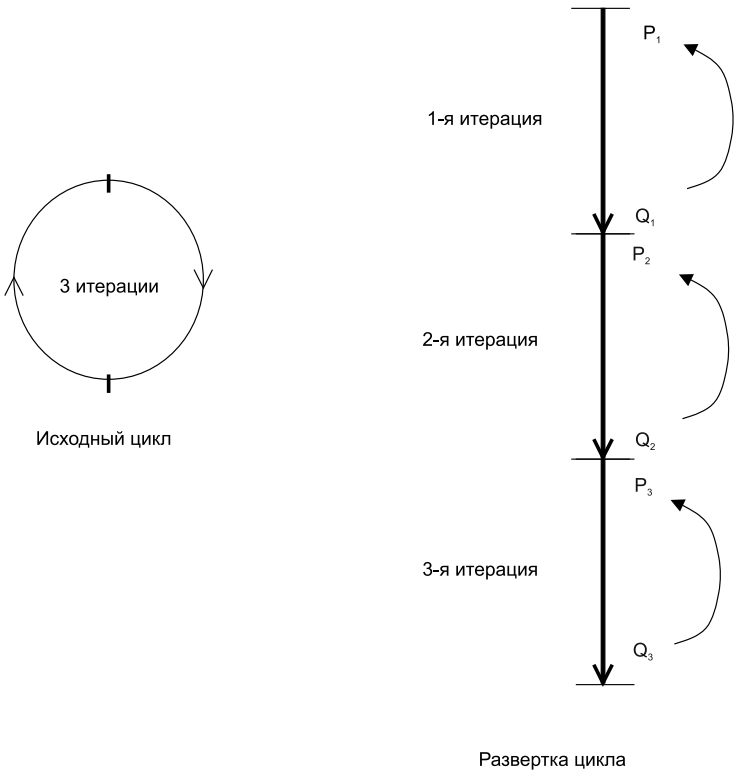


Рис. 7.5. Обратная развертка цикла FOR

Примем, что в теле цикла `FOR` используются только операторы присваивания. В этом случае каждая итерация развернутого цикла доказывается с помощью аксиомы присваивания и правила вывода для последовательности, обеспечивающих вычисление P_3 по Q_3, P_2 по Q_2 и P_1 по Q_1. Переход же от одной итерации к соседней также требует применения лишь правила вывода для последовательности, поскольку в нашем примере нужно только установить тождества: $Q_2 = P_3$ и $Q_1 = P_2$.

Отсюда заключение: самостоятельное правило вывода для оператора цикла FOR не требуется.

Правило вывода для оператора цикла WHILE

Для цикла `WHILE` количество итераций заранее неизвестно, поэтому применяется самостоятельное правило вывода, основанное на использовании инварианта:

$$\frac{\{I \ and \ B\} \ S \ \{I\}\}}{\{I\} \ while \ B \ do \ S \ end \ \{I \ and \ (not \ B)\}}'$$

где I — инвариант цикла.

Анализ правила:

1. Слабейшее предусловие должно иметь истинное значение до начала цикла и должно гарантировать истинность инварианта.
2. Постусловие должно иметь истинное значение после окончания цикла при истинном значении инварианта.
3. В ходе выполнения цикла истинность инварианта не должна зависеть от вычисления булева выражения, управляющего циклом, и от операторов тела цикла.
4. Инвариант должен иметь истинное значение до начала цикла, во время выполнения цикла и после завершения цикла. Отсюда и имя — инвариант.

Требования к инварианту цикла

`P` ⇒ `I`	(изначально инвариант цикла должен иметь истинное значение)
`{I} B {I}`	(вычисление `B` не должно влиять на `I`)
`{I and B} S {I}`	(выполнение тела цикла не должно влиять на `I`)
`(I and (not B))` ⇒ `Q`	(из `I = true` и `B = false` следует `Q`)
Цикл завершается	(иногда это трудно доказать)

Определение инварианта цикла по индукции

Определяют инвариант по индукции, многократно вычисляя слабейшие предусловия. При этом полагают, что на основе частных случаев будет выявлено общее утверждение.

Выработку слабейшего предусловия удобно рассматривать как вычисление псевдофункции wp:

```
wp (оператор_тела, постусловие) = предусловие
```

Для нахождения значений `wp` выполняем обратную развертку итераций тела цикла, начиная с нуля итераций (рис. 7.6).

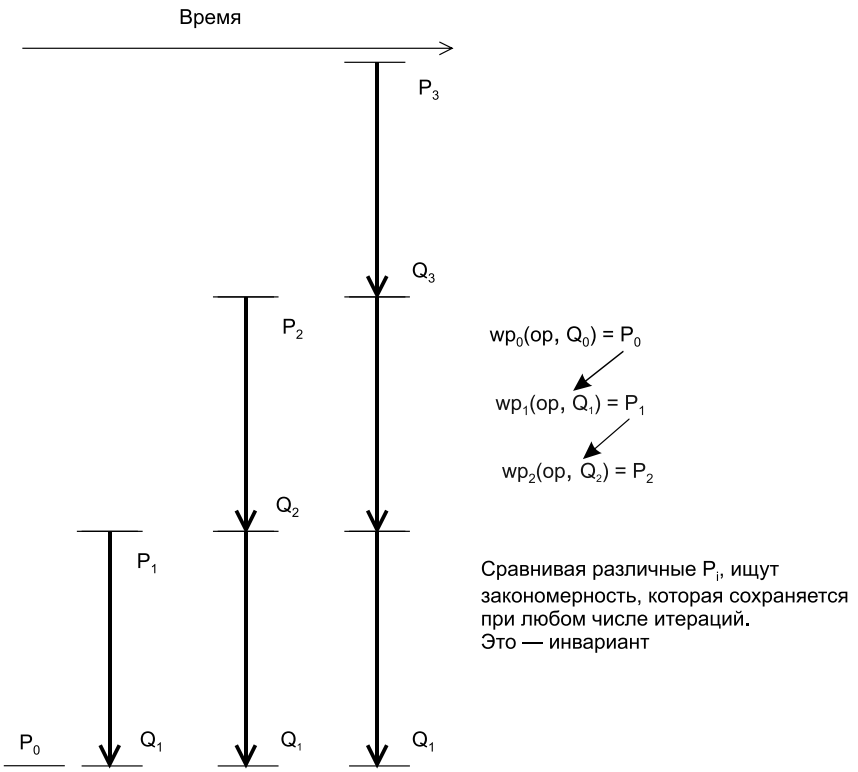


Рис. 7.6. Обратная развертка цикла WHILE

Если тело цикла содержит единственный оператор присваивания, то инструментами вычисления будут:

1. Аксиома присваивания (для перехода в рамках одной итерации от постусловия к предусловию).
2. Правило вывода для последовательности (для перехода от предусловия к постусловию соседних итераций).

Пример доказательства цикла

Рассмотрим цикл: `while y <> x do y := y + 1 end {y = x}`

Построение инварианта

Очевидно, что для нуля итераций слабейшее предусловие: `{y = x}`.

Для одной итерации:

```
wp (y := y + 1, {y = x}) = {y + 1 = x} = {y = x - 1}
```

Для двух итераций:

```
wp (y := y + 1, {y = x - 1}) = {y + 1 = x - 1} = {y = x - 2}
```

Для трех итераций:

```
wp (y := y + 1, {y = x - 2}) = {y + 1 = x - 2} = {y = x - 3}
```

Ясно, что `{y < x}` удовлетворяет условиям одной и более итераций.

Объединяя это утверждение с `{y = x}` для случая нуля итераций, получаем инвариант цикла:

```
I = {y < x} ∪ {y = x} = {y <= x}
```

Его можно использовать и как предусловие, то есть `P = I`.

Конечно, мы должны быть уверены, что наш инвариант удовлетворяет всем пяти требованиям.

Проверка требований к инварианту I

1. Поскольку `P = I`, `P => I`.
2. Очевидно, что `I` не зависит от `y <> x`, которое является условием продолжения цикла.
3. Далее, должна обеспечиваться истинность `{I and B} S {I}`. В нашем примере имеем

   ```
   {(y <= x) and (y <> x)} y := y + 1 {y <= x}
   ```

   Применяя аксиому присваивания к `y := y + 1 {y <= x}`,
   получим `{y + 1 <= x}` ⇔ `{y < x}`.
   Поскольку `{(y <= x) and (y <> x)} => {y < x}`, утверждение доказано.
4. Теперь следует доказать: `{(y <= x) and not (y <> x)} => {y = x}`.
   Раскроем исходную посылку: `{(y <= x) and (y = x)} => {y = x}`.
   Получаем: `{y = x} => {y = x}`. Таким образом, и это доказано.
5. В заключение надо решить вопрос о завершении нашего цикла

   ```
   {y <= x} while y <> x do y := y + 1 end {y = x}
   ```

   Учитывая, что `x` и `y` — целые числа, легко видеть: цикл завершается за конечное число шагов. Этот шаг завершает доказательство корректности полученного инварианта.

Если выполнение 5-го требования (завершение цикла) не гарантируется, то описание цикла называют *частично корректным*, в противном случае — *полностью корректным*.

Общий случай определения инварианта цикла

В более сложных циклах нахождение инварианта, даже для частичной корректности, требует большой изобретательности. Так как вычисление предусловия для

цикла while зависит от нахождения инварианта, аксиоматическое доказательство корректности программ с циклами while может быть очень трудным.

Рассмотрим пример доказательства программы, вычисляющей функцию факториала:

```
{n > 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

Метод нахождения инварианта по индукции здесь не работает. Требуется некоторая изобретательность.

Здесь переменная count последовательно принимает значения n, n–1, …, 1, 0 (count означает: сколько операций умножения остается выполнить для того, чтобы получить n!). Переменная fact последовательно принимает значения

```
n, n * (n - 1), n * (n - 1) * (n - 2), …, n * (n - 1) * …* 2 * 1
```

Цикл вычисляет функцию так, что последнее умножение выполняется первым. Это значит, что первым вычисляется (n - 1) * n при допущении, что n > 1.

Итак, частью инварианта может быть

```
fact = (count + 1) * (count + 2) * …* (n - 1) * n
```

Но мы должны быть уверены, что величина count всегда не отрицательная, поэтому добавим это условие и получим инвариант:

```
I = (fact = (count + 1) * …* n) and (count >= 0)
```

Далее, мы должны проверить, выполняет ли этот инвариант пять требований.

Первое требование. Опять положим, что I будет использоваться как предусловие (это значит, что первое требование соблюдается).

Второе требование. Ясно, что вычисление булева условия count <> 0 не влияет на I (тем самым подтвердили второе требование).

Третье требование. Следующий вопрос: истинность {I and B} S {I}.

```
I and B – это ((fact = (count + 1) * …* n) and (count >= 0)) and (count <> 0)
```

Это предложение редуцируется до (fact = (count + 1) * …* n) and (count > 0).

В нашем примере нужно вычислить предусловие тела цикла, используя инвариант как постусловие.

Для {P} count = count - 1 {I} вычисляемое P примет вид:

```
{(fact = count * (count + 1) * …* n) and (count >= 1)}
```

Используем его как постусловие для 1-го присваивания в теле цикла:

```
{P} fact = fact * count {(fact = count * (count + 1) * …* n) and (count >= 1)}
```

В этом случае P равно:

```
{(fact = (count + 1) * …* n) and (count >= 1)}
```

Сравним это значение со значением I and B. Поскольку предусловие всегда можно усилить, третье требование исполнено.

Четвертое требование. Наконец, четвертая проверка касается утверждения:

```
(I and (not B)) ⇒ Q
```

Для нашего примера это означает:

```
((fact = (count + 1) * …* n) and (count >= 0)) and (count = 0)) ⇒ fact = n!
```

Совершенно ясно, что при `count = 0` левая часть является точным определением факториала.

Следовательно, созданное высказывание `I` признается истинным инвариантом.

Теперь можно использовать утверждение `P` (то же, что и `I`) из `while`-оператора как постусловие для второго оператора присваивания в программе:

```
{P} fact = 1 {(fact = (count + 1) * …* n) and (count >= 0)}
```

и сформировать утверждение для предусловия 2-го оператора:

```
(1 = (count + 1) * …* n) and (count >= 0)
```

Используя его как постусловие для 1-го оператора присваивания в программе

```
{P} count = n {(1 = (count + 1) * …* n) and (count >= 0)}
```

получим для предусловия *P* 1-го оператора:

```
{(n + 1) * …* n = 1) and (n >= 0)}
```

Левый операнд логического умножения равен истине (так как `1 = 1`), а правый операнд точно совпадает с предусловием всей программы `{n > 0}`. Таким образом, правильность программы доказана.

Денотационная семантика

В этом подходе пытаются синтезировать определение функции, которую вычисляет каждая программа, написанная на языке программирования. Спецификация языка создается как иерархия определений функций, вычисляемых каждой отдельной программной конструкцией.

В программе любая операция представляет собой некоторую математическую функцию. Для композиции этих функций в более крупные фрагменты используются выражения и операторы. В свою очередь, линейные последовательности операторов и условные ветвления также могут быть представлены функциями, составленными из функций отдельных компонентов этих конструкций. Цикл легко описывается рекурсивной функцией, составленной из компонентов, входящих в его тело. В конечном счете образуется функциональная модель всей программы. Примерами такого подхода к заданию семантики являются метод денотационной семантики Скотта и Стрэчи, а также метод функциональной семантики Миллза [31–33, 83].

Денотационная семантика — наиболее строгий метод для описания смысла программ. Он основан на теории рекурсивных функций.

Здесь для каждой конструкции языка задается:

1) математический объект;
2) семантическая функция, которая отображает экземпляры конструкции в экземпляры математического объекта.

Поскольку математические объекты определены точно, они представляют точный смысл синтаксических конструкций языка.

Чаще всего в качестве математического объекта используется множество положительных целых чисел *N*. Поэтому смысл языковых конструкций определяется значениями математических целых чисел.

Задать денотационную семантику языка — это значит определить семантические функции отображения для всех его понятий. Название «денотационная семантика» происходит от слова *denote* (обозначать), так как математический объект обозначает смысл соответствующей синтаксической единицы.

Семантическая функция отображения двоичных чисел

Простейшая языковая конструкция — двоичные числа.

Синтаксис двоичного числа описывается следующими правилами подстановки:

```
<bin_num> ::= 0|1|<bin_num> 0|<bin_num> 1
```

Смысл двоичных чисел представляется *математическими* десятичными числами, вычисляемыми с помощью функции M_{bin}:

```
M_bin ('0') = 0
M_bin ('1') = 1
M_bin (<bin_num> '0') = 2 * M_bin (<bin_num>)
M_bin (<bin_num> '1') = 2 * M_bin (<bin_num>) + 1
```

ПРИМЕЧАНИЕ

Синтаксические цифры помещены в апострофы, чтобы отличать их от математических цифр.

Математические значения определяют только два первых правила. Остальные два являются по сути правилами промежуточных вычислений.

Примеры вычисления семантических значений

```
M_bin ('10') = 2 * M_bin ('1') = 2 * 1 = 2
M_bin ('01') = 2 * M_bin ('0') + 1 = 2 * 0 + 1 = 1
M_bin ('11') = 2 * M_bin ('1') + 1 = 2 * 1 + 1 = 3
M_bin ('110') = 2 * M_bin ('11') = 2 * (2 * M_bin '1') + 1) = 2 * (2 * 1 + 1) = 6
```

Семантическая функция отображения десятичных чисел

Следующая языковая конструкция — десятичные числа.

Синтаксис десятичного числа описывается следующими правилами подстановки

```
<dec_num> ::= 0|1|2|3|4|5|6|7|8|9
              |<dec_num> (0|1|2|3|4|5|6|7|8|9)
```

Смысл десятичных чисел представляется математическими десятичными числами, вычисляемыми с помощью функции M_{dec} :

```
M_dec ('0') = 0, M_dec ('1') = 1, M_dec ('2') = 2,…, M_dec ('9') = 9
M_dec (<dec_num > '0') = 10 * M_dec (<dec_num >)
M_dec (<dec_num > '1') = 10 * M_dec (<dec_num >) + 1
. . .
M_dec (<dec_num > '9') = 10 * M_dec (<dec_num >) + 9
```

Примеры вычисления семантических значений

M_{dec} ('10') = 10 * M_{dec} ('1') = 10 * 1 = 10
M_{dec} ('01') = 10 * M_{dec} ('0') + 1 = 10 * 0 + 1 = 1
M_{dec} ('08') = 10 * M_{dec} ('0') + 8 = 10 * 0 + 8 = 8
M_{dec} ('11') = 10 * M_{dec} ('1') + 1 = 10 * 1 + 1 = 11
M_{dec} ('25') = 10 * M_{dec} ('2') + 5 = 10 * 2 + 5 = 25
M_{dec} ('110') = 10 * M_{dec} ('11') = 10 * (10 * M_{dec} ('1') + 1) =
=10 * (10 * 1 + 1) = 110
M_{dec} ('457') = 10 * M_{dec} ('45') + 7 = 10 * (10 * M_{dec} ('4') + 5) + 7 =
=10 * (10 * 4 + 5) + 7 = 457

Состояние программы

Денотационная семантика программы определяется в терминах изменений состояния идеального компьютера.

Ключевая разница между операционной и денотационной семантикой состоит в том, что в операционной семантике изменения состояния определяются кодируемыми алгоритмами, а в денотационной семантике изменения состояния определяются строгими математическими функциями.

Пусть состояние S программы представляется набором упорядоченных пар элементов:

S = {<i_1, v_1>, <i_2, v_2>, …, <i_n, v_n>}

где i_j — имя j-й переменной программы; v_j — текущее значение этой переменной.

Положим, что в некоторой программе имеются три переменные с именами a, b, c и значениями 1, 10 и 50 соответственно.

Как показано на рис. 7.7, для хранения этой информации можно использовать «двухэтажную» структуру из шести ячеек памяти:

i	a	b	c
v	1	10	50

Рис. 7.7. Состояние программы с тремя переменными

В ячейках «верхнего этажа» (это ячейки i_1, i_2, i_3) хранят имена переменных, а в ячейках «нижнего этажа» (это ячейки v_1, v_2, v_3) — значения переменных.

Любой элемент v может иметь специальное значение *undef*, указывающее, что связанная с ним переменная не определена.

Опросить состояние программы позволяет функция VARMAP (i_j, S).

Эта функция для данного имени переменной i_j и состояния S возвращает текущее значение переменной:

VARMAP (i_j, S) = v_j

Для нашего примера:

VARMAP (i_1, S) = 1, VARMAP (i_2, S) = 10, VARMAP (i_3, S) = 50

Большинство семантических функций для программ и программных конструкций отображают состояния в состояния. Эти изменения состояний используются для определения смысла программных конструкций.

Такие языковые конструкции, как выражения, отображаются в значения, а не в состояния.

Выражения

Будем рассматривать очень простые выражения:

- в них используется только одна операция + или *;
- операндами являются только целые переменные и константы;
- скобки не используются;
- значение выражения — целое число.

BNF-правила для таких выражений имеют вид:

```
<expr> ::= <dec_num>|<var>|<bin_expr>
<bin_expr> ::= <left_expr> <op> <right_expr>
<op> ::= +|*
```

Будем учитывать лишь одну ошибку выражения — результат имеет неопределенное значение error. Тогда набор значений, которые может дать выражение, равен $N \cup \{\texttt{error}\}$.

Определим отображающую функцию для некоторого выражения *E* и состояния *S*.

Условимся использовать для записи семантических функций символ Δ = (чтобы отличать их от операторов присваивания).

```
Me (<expr>, S) Δ =
    case <expr> of
        <dec_num> ⇒ Mdec (<dec_num>, S)
        <var> ⇒ if VARMAP (<var>, S) = undef
            then error -- или undef, если это рекурсивный вызов
            else VARMAP (<var>, S)
        <binary_expr> ⇒
            if (Me (<binary_expr>.<left_expr>, S) = undef OR
                Me (<binary_expr>.<right_expr>, S) = undef)
                then error
                else if (<binary_expr>.<op> = '+') then
                    Me (<binary_expr>.<left_expr>, S) +
                        Me (<binary_expr>.<right_expr>, S)
                else Me (<binary_expr>.<left_expr>, S) *
                        Me (<binary_expr>.<right_expr>, S)
```

Как видим, здесь заданы три направления вычислений:

1. *Выражение является десятичным числом*. В этом случае всю работу выполняет семантическая функция M_{dec}.
2. *Выражение является переменной*. Проверяется: имеет ли переменная значение; после чего значение возвращается из состояния программы S с помощью функции VARMAP. Если же значение не определено, фиксируется ошибка.
3. *Выражение содержит операцию*, которая применяется к значениям аргументов, вычисляемых рекурсивными вызовами функции M_e.

ПРИМЕЧАНИЕ

Точное описание вычислительного процесса, порождаемого применением отображающей функции, не может быть представлено средствами императивного ЯП; оно задается средствами λ-исчисления.

ПРИМЕЧАНИЕ

Если функция вызвана рекурсивно, то она не имеет права сгенерировать ошибку (error), а возвращает значение undef, которое приводит к генерации ошибки.

Операторы присваивания

По сути, выполнение оператора присваивания складывается из вычисления выражения (из правой части) и занесения в переменную (из левой части) значения выражения. Его смысл можно описать следующей функцией:

```
M_a (x = E, S) Δ =
       if  M_e (E, S) = error
           then error
           else S' = {<i'_1, v'_1>, <i'_2, v'_2>, …, <i'_n, v'_n>},
               где для j = 1, 2, …, n:
                   при i_j < > x     v'_j = VARMAP (i_j, S);
                   при i_j = x       v'_j = M_e (E, S).
```

ПРИМЕЧАНИЕ

S' — новое состояние программы, в нем изменилось значение только одной переменной x.

В двух последних строках используются сравнения имен i_j<> x и i_j = x, а не значений.

Логические циклы с предусловием

Будем считать, что в нашем распоряжении имеются:

- M_{sl} — функция отображения списка операторов из тела цикла в состояние;
- M_b — функция отображения булева выражения в булеву величину.

Тогда семантическую отображающую функцию для цикла `while` можно описать в следующем виде

```
M_w (while B do L, S) Δ =
    if M_b (B, S) = undef
        then error
        else  if M_b (B, S) = false
            then S
            else if M_sl (L, S) = error
                then error
                else M_w (while B do L, M_sl (L, S))
```

При отсутствии ошибок значение цикла — это значения программных переменных после выполнения в цикле операторов, причем заданное число раз.

По сути, здесь цикл преобразуется из итерации в рекурсию. Необходимость преобразования диктуется простым соображением: математически рекурсию

описать легче, чем итерацию. Для управления количеством рекурсивных вызовов используются вспомогательные семантические функции.

Контрольные вопросы и упражнения

1. Для синтаксически управляемого определения из табл. 7.1 постройте аннотированные деревья разбора следующих выражений:

```
а) (3 + 4) * (5 + 6);
б) 1 * 2 * 3 * (4 + 5);
в) (9 + 8 * (7 + 6) + 5) * 4.
```

2. Используя команды виртуальной машины, дайте операционное семантическое определение:

   а) для оператора цикла языка С `for ( i = 1; i <= N; i=i+1) S`;
   б) для цикла `repeat` для языка Паскаль (допустим, что логическое выражение является простым выражением отношения) `repeat...until <выражение_отношения>`;
   в) для цикла `for-downto` языка Паскаль `for i := first downto last do...`;
   г) для цикла языка Фортран `DO k=start, end, step <действия> END DO`;
   д) для цикла for языка Си `for (expr1; expr2, expr3) вычислить expr4`.

3. Вычислить слабейшее предусловие для оператора присваивания и постусловия.

   Дано:
   оператор присваивания: `a := 2 * (b – 1) – 1`
   постусловие: `{a > 0}`
   Дано:
   оператор присваивания: `b := (c + 10) / 3`
   постусловие: `{b > 6}`
   Дано:
   оператор присваивания: `a := a + 2 * b – 1`
   постусловие: `{a > 1}`
   Дано:
   оператор присваивания: `x := 2 * y + x – 1`
   постусловие: `{x > 11}`

4. Вычислить слабейшее предусловие для последовательности операторов присваивания и ее постусловия.

   Дано:
   оператор присваивания: `a := 2 * b + 1`
   оператор присваивания: `b := a – 3`
   постусловие: `{b < 0}`
   Дано:
   оператор присваивания: `a := 3 * (2 * b + a)`
   оператор присваивания: `b := 2 * a – 1`
   постусловие: `{b > 5}`

5. Доказать, что следующая программа корректна:

```
{y = V_y and x = V_x}
temp = x;
x = y;
y = temp;
{x = V_y and y = V_x}
```

6. Доказать аксиоматическую корректность следующего оператора цикла:

```
{x = 3, y = x - 3} for i := 1 to 3 do y := y + 1 {x = 3, y = x}
```

7. Доказать, что следующая программа корректна:

```
{n > 0}
count = n;
sum = 0;
    while count <> 0 do
        sum = sum + count;
        count = count - 1;
    end
{sum = 1 + 2 + … + n}
```

Решение

Проведем следующие рассуждения. Переменная `count` здесь последовательно принимает значения n, $n-1$, ..., 1.

Переменная `sum` последовательно принимает значения n, $n+(n-1)$, $n+(n-1)$, $n+(n-1)+(n-2)$, ..., причем последнее значение равно сумме членов арифметической прогрессии $\frac{(1+n)*n}{2}$.

Очевидно, что текущее значение *sum* отличается от конечного значения на величину $\frac{(1+count)*count}{2}$. Эта величина равна сумме членов ряда арифметической прогрессии для элементов от *count* до 1, то есть еще не учтенных при вычислении элементов. Поэтому в каждый момент времени справедливо:

$$sum+\frac{(1+count)*count}{2}=\frac{(1+n)*n}{2}.$$

Это утверждение может быть положено в основу инварианта цикла. Дополнительно требуется проверять положительность величины *count*. Поэтому полное утверждение для инварианта принимает вид:

$$I=(sum+\frac{(1+count)*count}{2}=\frac{(1+n)*n}{2}) \quad and \quad (count\geq 0).$$

Проверим выполнение требований к инварианту.

Первое требование. Примем, что $P = I$, поэтому выполняется $P \Rightarrow I$.

Второе требование. Вычисление условия цикла *count* <> 0 не влияет на I.

Третье требование. Надо доказать, что выполнение тела цикла не влияет на I: $\{I \text{ and } B\}\ S\ \{I\}$.

$$I \quad and \quad B=$$

$$=((sum+\frac{(1+count)*count}{2})=\frac{(1+n)*n}{2}) \quad and \quad (count\geq 0)) \quad and \quad (count\neq 0)=$$

$$=((sum+\frac{(1+count)*count}{2})=\frac{(1+n)*n}{2}) \quad and \quad (count>0).$$

Мы должны вычислить предусловие тела цикла, используя инвариант как постусловие.
Для последнего оператора тела
$\{P_2\}\ \ count = count - 1\ \ \{I\}$

предусловие равно

$$\{(sum + \frac{count * (count - 1)}{2} = \frac{(1+n) * n)}{2})\ \ and\ \ (count \geq 1)\}.$$

Используем это утверждение как постусловие для 1-го оператора тела

$$\{P_1\}\ \ sum = sum + count\ \ \{(sum + \frac{count * (count - 1)}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (count \geq 1)\}.$$

Предусловие для этого оператора принимает вид

$$\{(sum + \frac{count * (1 + count)}{2} = \frac{(1+n) * n)}{2})\ \ and\ \ (count \geq 1)\}.$$

Ясно, что оно равно I and B, то есть I and $B \Rightarrow P_1$.
Четвертое требование: (I and (not B)) ⇒ Q.
В нашем примере:

$$I\ \ and\ \ (not\ \ B) =$$

$$= (sum + \frac{(1 + count) * count}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (count \geq 0)\ \ and\ \ (count = 0) =$$

$$= (sum = \frac{(1+n) * n}{2})\ \ and\ \ (count = 0) = Q.$$

Пятое требование: цикл должен завершаться. Количество повторений цикла задает значение переменной *count*. При каждом повторении тела величина *count* уменьшается — значит, цикл завершается.
Теперь используем предусловие для `while`-оператора как постусловие для оператора `sum = 0` программы:

$$\{P\}\ \ sum = 0\ \ \{(sum + \frac{(1 + count) * count}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (count \geq 0)\}.$$

Получим предусловие для этого оператора:

$$\{(\frac{(1 + count) * count}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (count \geq 0)\}.$$

Используем это утверждение как постусловие для оператора `count = n`:

$$\{P\}\ \ count = n\ \ \{(\frac{(1 + count) * count}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (count \geq 0)\}.$$

Получим предусловие для этого оператора `count = n`:

$$\{(\frac{(1+n) * n}{2} = \frac{(1+n) * n}{2})\ \ and\ \ (n \geq 0)\} = \{n \geq 0\}.$$

Поскольку по правилу консеквенции предусловие может быть усилено ($n > 0$) ⇒ ($n \geq 0$), правильность программы доказана.

8. Записать отображающую функцию денотационной семантики для следующих конструкций:

а) оператор `for` для языка Pascal

```
M_pf (for var := init_expr to final_expr do L, S) Δ =
    if VARMAP (i, s) = undef  -- для i   init_expr  или final_expr
        then error
        else if M_e (init_expr, S) > M_e (final_expr, S)
            then S
            else M_w (while init_expr – 1 <= final_expr do L,
                M_a (var := init_expr + 1, S))
```

б) оператор `repeat` для языка Pascal

```
M_r (repeat L until B) Δ =
    if M_b (B, S) = undef
        then error
        else  if M_sL (L, S) = error
            then error
            else if M_b (B, S) = true
                then M_s1 (L, S)
                else M_r (repeat L until B), M_sL (L, S))
```

в) булево выражение для языка Pascal

```
M_b (B, S) Δ =if VARMAP (i, S) = undef  -- для некоторой i в B
    then error
    else B', -- где B' - результат вычисления B после установки
             -- каждой переменной i в B с помощью VARMAP(i, S)
```

г) оператор `for` для языка Си

```
M_cf (for (expr1, expr2, expr3) L, S) Δ =
    if VARMAP (i, S) = undef  -- для некоторой i в expr1, expr2, expr3 или L
        then error
        else if M_e (expr2, M_e (expr1, S) = 0
            then S
            else M_help (expr2,expr3, L, S)
M_help (expr2, expr3, L, S) Δ =
    if VARMAP (i, S) = undef  -- для некоторой i  в expr2, expr3 или L
        then error
        else  if M_sL (L, S) = error
            then S
            else M_help (expr2, expr3, L, M_sL (L, M_e (expr3, S))
```

Глава 8
Типизация данных

Восьмая глава вводит в круг вопросов, связанных с понятиями объект и тип данных. В этой главе рассматриваются: объекты данных, переменные и константы как разновидности объектов, типы данных в качестве инструмента классификации и контроля объектов, объявления как описания объектов в программе. Поясняется концепция механизмов статического и динамического контроля типов. Последовательно обсуждаются шесть атрибутов переменной, в состав которых входят: имя, тип, адрес, значение, время жизни, область видимости. Исследуется статическое и динамическое связывание атрибутов переменной. Значительное внимание уделяется классификации переменных по месту размещения и времени жизни. Кроме того, здесь определяются и объясняются понятия «система типизации данных» и «тип выражения».

Объекты данных

Любая программа несет в себе набор операций, которые применяются к определенным данным в определенной последовательности. Каждый язык программирования задает три категории характеристик:

1) допустимые значения данных и способы размещения значений в памяти компьютера;
2) допустимые операции (встроенные в язык и рукотворные, то есть создаваемые программистом);
3) операторы, управляющие последовательностью применения операций к данным.

Области хранения данных в аппаратуре компьютера (памяти, регистрах и внешних запоминающих устройствах) имеют довольно простую структуру, которая группирует последовательности битов информации в байты или слова. Однако в вычислительной среде предусматриваются более сложные формы хранения данных: стеки, массивы, числа, символьные строки и т. д. Элементы данных, рассматриваемые как единое целое в некий момент выполнения программы, обычно называют *объектами данных*. В ходе вычислений существует великое множество различных объектов данных. Мало того, объекты данных и отношения между ними динамически меняются в разные моменты вычислительного процесса.

Одни объекты данных определяются программистом (переменные, константы, массивы, файлы и т. п.). Программист создает эти объекты и явно управляет ими (с помощью объявлений и операторов в программе). Другие объекты данных самостоятельно создаются компьютерной системой. К таким объектам данных (стекам, записям активации процедур и функций, файловым буферам и спискам свободной памяти) программист непосредственного доступа не имеет. Подобные объекты формируются системой автоматически и обеспечивают «машинную» организацию вычислений.

Объект данных является *контейнером*, содержащим значения данных и другую информацию о данных. В этом контейнере значения хранятся и из него они извлекаются. Объект данных представляет собой сложную конструкцию, имеющую многочисленный набор атрибутов. Самым важным атрибутом считается *тип данных*. В целом, атрибуты определяют специфику допустимых значений, логическую организацию этих значений и особенности работы с ними.

Значение данных может быть отдельным числом, символом или указателем на другой объект данных. Когда говорят о значении объекта данных, имеют в виду следующее: в области памяти компьютера, выделенной объекту, размещается некоторая комбинация битов, соответствующая значению.

В ходе вычислений одни объекты данных существуют с самого начала выполнения программы, а другие создаются динамически в какие-то моменты времени. Одни объекты данных уничтожаются на определенных этапах вычислений, а другие сохраняются до конца работы программы. Следовательно, у каждого объекта данных свой период жизни, в течение которого он может быть задействован для хранения значений данных. Объект данных считается элементарным, если содержащееся в нем значение всегда рассматривается как единое целое. Если же этот объект состоит из набора других объектов, его называют структурой данных.

Любой объект данных в течение своей жизни подвергается изменениям. Хотя состав атрибутов объекта данных не меняется, этого нельзя сказать об их содержании. Приведем самые важные атрибуты (характеристики) объекта данных:

- *Тип*. Этот атрибут задает возможные значения, применимые операции и формат хранения данных.
- *Местоположение*. Данный атрибут хранит координаты области памяти, отведенной под объект данных. Конкретная область памяти назначается системными программными средствами, управляющими памятью. Программист обычно не имеет доступа к этим средствам.
- *Значение*. Это текущая величина (или набор величин) объекта данных.
- *Имя*. Перечень имен, по которым к объекту данных обращаются в процессе вычислений. Имена могут назначаться как статически (в начале работы), так и динамически, в определенные моменты времени вычислений.
- *Принадлежность*. Сведения о принадлежности данного объекта к другим объектам данных (например, ссылки на эти объекты).

Переменные и константы

Переменная является объектом данных, который программист явно определяет и называет. Имя переменной считают нечувствительным к регистру, если не имеет

значения, какими буквами оно записано (заглавными или строчными). Впрочем, во многих языках имена `переменная` и `ПЕРЕМЕННАЯ` обозначают разные объекты данных. В течение жизни значение переменной может многократно изменяться. Говорят, что при этом происходит новое связывание переменной с ее значением.

Константой называют объект данных с неизменным значением. Различают обычные и именованные константы. Для обычной константы задается только значение. В именованной константе значение связывается с явно определяемым именем (это связывание происходит только один раз).

Например, программа на языке C может содержать следующее объявление:

```
int X;
```

что означает описание простой целой переменной с именем `X`. По этому объявлению система создает объект данных, выделяя под него память. Изначально этот объект значения не имеет. Далее может быть записан оператор присваивания `X = 15`, связывающий объект данных `X` со значением `15`.

Для определения константы можно написать объявление

```
const int ARG=45;
```

Здесь указывается, что в жизненном цикле программы объект данных с именем `ARG` постоянно должен быть связан со значением `45`.

Предусмотрен и другой способ объявления константы — макроопределение:

```
#define ARG 45
```

Макроопределение обрабатывается во время компиляции программы и приводит к тому, что все ссылки на имя `ARG` в программе будут заменены на константу `45`.

Заметим, что у константы `45` имеются два имени: определенное программистом имя `ARG` и буквальное имя `45`. Оба эти имени могут быть использованы в программе для ссылки на один и тот же объект данных со значением 45. Имя `45` считается *литералом*, который именует объект данных, содержащий значение `45`. Способ создания и использования литералов определяется самим языком программирования.

Важно понимать тонкое отличие значения `45`, являющегося целым числом, которое хранится в памяти компьютера во время выполнения программы, от имени `45`, состоящего из двух символов, `4` и `5`, и представляющего собой десятичную форму записи в тексте программы того же числа.

Следует знать, что макроопределение дает команду компилятору приравнять `ARG` значению `45`, а служебное слово `const` указывает компилятору, что переменная `ARG` всегда содержит значение `45`.

Типы данных

Физический мир, являющийся предметной областью программы, содержит разнообразные объекты, рассматриваемые в задачах вычислений как элементы данных. Это могут быть дома, машины, деревья, самолеты, собаки, люди, туфли и т. д. Все эти физические объекты отличаются друг от друга, имеют разнообразные характеристики. Для удобства вычислений физические объекты можно классифицировать, выделяя, например, класс домов, класс автомобилей. Программы же обрабатывают представления элементов данных.

Представления элементов данных в императивных языках программирования создаются с помощью объектов данных. Различные категории (классы) объектов характеризуются типами данных.

Тип данных — это механизм классификации объектов данных [12, 17, 42, 56, 65, 80, 83, 86, 88].

Тип данных определяет:

1) значения объектов;
2) операции, применимые к значениям;
3) размещение значений в машинной памяти.

Иными словами, тип данных — это двойка

```
<Тип_данных> =< <Алгебра_операций>, <Формат_размещения_значений> >
```

где

```
<Алгебра_операций> = (<Значения>, <Операции>)
```

В каждом языке имеется некий набор встроенных примитивных типов данных. Дополнительно в языке предусматриваются средства, позволяющие программисту создавать новые типы данных. Одно из главных различий между ранними языками программирования (Fortran, Cobol) и более поздними (C++, C#, Java, Ada) относится к типам данных, определяемым программистом. Современный подход заключается в том, что язык *должен* предоставлять средства для расширения набора типов данных.

Основными элементами описания типа данных являются:

- *служебные атрибуты*, которые характеризуют объекты данного типа;
- *значения*, которые могут принимать объекты данного типа;
- *операции*, которые определяют возможные действия над объектами данного типа.

Например, при описании типа данных «массив» в состав служебных атрибутов входят:

- количество размерностей массива;
- допустимый диапазон изменения индекса для каждой размерности;
- тип данных отдельного элемента массива.

Значения могут задаваться перечнем чисел (диапазоном или перечислением), которые фиксируют допустимые значения элементов массива.

Набор возможных операций для типа «массив» образуют:

- операция индексного доступа к элементам массива (каждому элементу приписывается определенный индекс);
- операция по созданию массива;
- операция изменения формы массива;
- операции запроса атрибутов массива (верхней и нижней границ диапазона индексов, размера);
- операция присвоения значения массива;
- арифметические операции над парами массивов.

При реализации типа данных обычно рассматриваются:

- *способ представления* объектов данных этого типа в памяти компьютера (в процессе вычислений);
- *способ представления операций*, определенных для этого типа данных. Он задает конкретные алгоритмы и процедуры для обработки выбранной формы размещения объектов данных в памяти.

Реализация типа данных определяет, каким образом данные виртуальной среды вычислений отображаются на аппаратные средства компьютера. Здесь может использоваться как программное, так и аппаратное моделирование.

Кроме того, при описании типа данных следует определить его синтаксис, то есть форму записи типа в тексте программы. И описание (спецификация), и реализация достаточно слабо зависят от конкретных синтаксических форм, применяемых в конкретном языке. Как правило, синтаксис атрибутов типа данных определяется формой объявления типа, принятой в языке. Значения могут представляться литералами или именованными константами. Вызов операций задается при помощи специальных символов, встроенных процедур или функций. Синтаксическое объявление обрабатывается компилятором, который в данном случае решает две задачи:

- максимально эффективно разместить данные в памяти;
- проверить соответствие данных указанному типу.

Каждая переменная (константа) считается экземпляром типа. Тип данных является важнейшим элементом аппарата прогноза и контроля языка. Например, именно знание типа переменной позволяет решать следующие вопросы:

- можно или нельзя присвоить переменной конкретное значение;
- можно или нельзя применить к переменной какую-то операцию.

В частности, запрещается заносить в переменную целого типа вещественные значения и применять к ней операцию конкатенации (объединения), допустимую для строкового типа.

Типы делятся на элементарные и составные. Элементарные (базовые) типы имеют атомарные значения, значениями составных типов являются структуры.

Элементарные типы данных

Элементарный объект данных может содержать только одно значение. Тип таких объектов, для которых задан набор операций, называется *элементарным типом данных*. Конечно, в каждом языке программирования присутствует свой перечень элементарных типов данных, но вещественный и целый типы, а также перечисления и указатели имеются почти во всех языках.

Содержание атрибутов объекта данных находится в той части его контейнера, которая называется *дескриптором*. Основные атрибуты объекта данных элементарного типа остаются неизменными в течение всего времени его жизни. Это относится, например, к имени и типу. Они востребованы в любой момент вычислений и хранятся в дескрипторе всегда. Другие атрибуты нужны лишь на этапе компиляции, поэтому на этапе выполнения из дескриптора изымаются. За счет этого экономится память под объект.

Тип определяет допустимые значения, которые могут содержаться в объекте данных как экземпляре данного типа. Например, целочисленный тип задает всевозможные целые числа, которые могут быть значениями целочисленных объектов данных. В частности, язык C разрешает четыре типа целых чисел: `int`, `short`, `long` и `char`. Каждый из этих типов достаточно просто реализуется аппаратными средствами обычного компьютера и поддерживает разную точность вычислений. Тип `short` обеспечивает наименьшую разрядность числа, тип `long` — наибольшую разрядность, а тип `int` ориентирован на самый эффективный формат представления целых чисел в памяти компьютера. Формат для `int` может совпадать с форматом для `short` или `long`, а может использовать и некоторую промежуточную форму. Символы в языке C хранятся как 8-битные целые числа типа `char`, который принадлежит к семейству целочисленных типов.

Набор значений для элементарных целочисленных объектов данных является упорядоченной последовательностью с наименьшим и наибольшим элементом.

Набор операций для какого-то типа данных задает возможные действия с объектами этого типа. Эти операции могут быть элементарными, то есть быть встроенными в язык, являться частью его определения, или они могут определяться программистом в виде процедур и функций.

Охарактеризуем спецификации некоторых элементарных операций. Спецификации будем задавать в следующем формате:

```
символ операции: тип первого операнда × тип второго операнда → тип результата
```

где символ × обозначает прямое (декартово) произведение множеств, представляемых значениями типов для операндов.

Здесь подразумевается спецификация двухместной (иначе бинарной) операции. Обычно спецификацию называют *сигнатурой операции*. В языке C такая запись именуется *прототипом* функции.

1. Целочисленное сложение — это операция, операндами которой являются два целочисленных объекта данных, а результатом — также целочисленный объект данных, обычно содержащий значение, равное сумме значений операндов. Если обозначить имя целого типа как `integer`, то спецификация записывается следующим образом:

   ```
   + : integer × integer → integer
   ```

2. Операция проверки на равенство значений двух целочисленных объектов данных. Результатом выполнения этой операции является булево значение (истина или ложь). Если обозначить имя булева типа как `boolean`, то спецификация примет вид:

   ```
   = : integer × integer →boolean
   ```

3. Операция извлечения квадратного корня. Она применима к объектам данных вещественного типа (с именем `real`):

   ```
   sqrt : real → real
   ```

Операция моделирует математическую функцию: каждому входному набору допустимых операндов она сопоставляет однозначный результат. У каждой операции имеется *область определения* (множество допустимых значений операндов)

и *область значений* (множество возможных значений для результата). Действие операции определяет, каковы будут результаты для данного набора входных значений (операндов). Действие операции обычно задается с помощью алгоритма.

При определении операции могут возникнуть отдельные трудности:

- Для некоторых значений операндов невозможно определить действие операции. Например, операция извлечения квадратного корня не определена для отрицательных целых чисел.
- Неявные операнды. Обычно при обращении к операции в программе ей передается некоторый набор явных операндов. Однако в операции могут быть задействованы неявные операнды (например, глобальные переменные). В этих условиях нелегко выявить все данные, которые влияют на результат операции.
- Побочные эффекты (неявные результаты). Операция может не только выдавать явный результат, но и как-то модифицировать значения, хранящиеся в других объектах данных. Такие неявные результаты называются побочными эффектами. Например, функция не только возвращает результат, но и изменяет значения переданных ей параметров.

Реализацию элементарного типа данных определяют:

- способ представления объектов рассматриваемого типа в памяти компьютера;
- набор алгоритмов, положенных в основу заданных операций.

Способ представления элементарных типов данных сильно зависит от аппаратных средств компьютера. Например, для целочисленных и вещественных величин используется двоичное представление, которое обеспечивается аппаратурой компьютера.

Атрибуты элементарного типа данных, вносимые в дескриптор объекта данных, могут выбираться из двух соображений:

1. *Эффективность*. В этом случае решаются две задачи: минимизировать затраты памяти и повысить скорость вычислений. В силу этого компилятор заносит в дескриптор объекта минимальное количество атрибутов (как правило, только имя и тип), все остальные атрибуты типа рассматриваются лишь на этапе компиляции. Такой подход характерен для языка С.
2. *Гибкость*. В данном подходе доминируют такие показатели, как гибкость и надежность (контроль) вычислений. В дескрипторы заносится максимальное количество атрибутов. Расплатой становятся повышенные затраты памяти и понижение скорости вычислений, поскольку аппаратура компьютера работу с атрибутами не поддерживает и все обращения к ним приходится реализовывать программным путем. Этот подход применяется в функциональных и логических языках.

Способ представления объекта данных обычно описывается в терминах необходимого размера области памяти (количества необходимых слов, байтов или битов) и компоновки атрибутов и значений данных в пределах этой области. Обычно адрес первого слова или байта в такой области памяти и характеризует местоположение объекта данных.

Любая операция может быть реализована одним из трех способов:

- Непосредственно аппаратурой компьютера. Например, если целые числа хранятся в памяти в «аппаратных» форматах, тогда сложение и вычитание целых чисел выполняется с помощью арифметических операций, встроенных в процессор.

- Как процедура или функция. Например, операция извлечения квадратного корня, как правило, не поддерживается аппаратными средствами. Она может быть реализована как программируемая функция.
- Как встраиваемая последовательность кодов микропрограммного уровня. Например, как подпрограмма микропрограммного управления процессором.

Объявления

В ходе создания программы программист задает имя и тип каждого используемого объекта данных.

Явное объявление — это оператор программы, сообщающий компилятору сведения об именах и типах объектов данных, которые применяются в программе. Место размещения объявления в программе определяет время жизни соответствующего объекта.

Например, если в начале функции, написанной на языке C, размещено объявление `float x, y;`, то оно указывает: во время выполнения функции потребуется два объекта данных типа `float`, которым назначаются имена `x` и `y`. Время жизни объекта `x` и объекта `y` ограничено периодом работы функции. После завершения выполнения функции эти объекты уничтожаются.

Во многих языках допускаются также *неявные объявления*, или объявления по умолчанию. Они применяются лишь тогда, когда отсутствуют явные объявления. Например, в программе на языке Fortran можно использовать простую переменную с именем `NUMBER`, не объявляя ее явным образом. Компилятор по умолчанию считает, что эта переменная имеет целый тип INTEGER, так как ее имя начинается с буквы, входящей в перечень I, J, K, L, M, N. В противном случае переменной был бы приписан вещественный тип REAL.

Механизм неявных объявлений все еще далек от совершенства. Следует отметить, что неявные объявления могут быть пагубными для надежности программы, поскольку препятствуют выявлению на этапе компиляции различных опечаток или ошибок программиста. Скажем, в языке Fortran переменные, которые программистом были случайно оставлены необъявленными, получат типы по умолчанию и очень неожиданные атрибуты, что может привести к неявным и трудно распознаваемым ошибкам. Многие программисты, пишущие на языке Fortran, теперь не используют неявные объявления. В силу этого компилятор отказывается от неявного объявления переменных, что позволяет избежать потенциальных проблем со случайно необъявленными переменными.

Некоторые проблемы неявного объявления можно устранить, если использовать подход, разработанный в скриптовых языках. Здесь имена отдельных типов должны начинаться с конкретных специальных символов. Например, в языке Perl все переменные с именами, начинающимися с символа `$`, считаются скалярными величинами, которые могут быть строками или числами. Если имя начинается с символа `@`, то именуемая переменная является массивом; если имя начинается с символа `%`, то оно обозначает хешированную структуру. Подобный подход создает различные пространства имен для переменных с различными типами. В этом случае имена `@music` и `%music` абсолютно различны, поскольку принадлежат различным

пространствам имен. Более того, читатель программы всегда знает тип переменной, когда читает ее имя. Данный подход существенно отличается от подхода в языке Fortran, так как там допускаются как явные, так и неявные объявления, поэтому тип переменной не всегда определяется написанием ее имени.

Другая разновидность неявного объявления основывается на контексте. Иногда это называют *логическим выводом типа*. В простейшем случае контекст задает тип начального значения, присваиваемого переменной в операторе объявления. Например, в языке C# объявление переменной `var` включает в себя начальное значение, тип которого должен стать типом переменной. Рассмотрим следующие объявления

```
var sum = 0;
var total = 0.0;
var name = "Liza";
```

Здесь типами переменных `sum`, `total` и `name` становятся `int`, `float` и `string` соответственно. Имейте в виду, что это статически типизированные переменные, у них типы неизменны в течение всего периода жизни переменной.

В языке Visual Basic (начиная с девятой версии), а также в функциональных языках ML, Haskell, OCaml и F# применяется вывод типа. В этих функциональных языках контекст, в котором появляется имя, является базисом для определения ее типа.

В общем случае при объявлении объекта данных можно также указать:

- неизменное значение объекта (если это константа) или начальное значение (если объект является переменной);
- размещение внутри другого объекта-контейнера;
- конкретное местоположение в памяти;
- конкретную форму представления.

При объявлении операции прописывают ее сигнатуру (типы операндов и результата). В языке C эта информация помещается в заголовок описания функции, ее прототип. Например, прототип

```
float fun(int a, float a)
```

объявляет функцию `fun` со следующей сигнатурой:

```
fun: int × float → float
```

Впрочем, для встроенных в язык элементарных операций такие сведения не нужны. Компилятору они известны по умолчанию.

Объявления поддерживают решение следующих важных задач:

- *Размещение данных в памяти*. Если объявление содержит сведения о типе и атрибутах объекта данных, то компилятор оптимизирует представление этого объекта данных в памяти.
- *Оптимизация управления памятью*. Содержащаяся в объявлении информация о времени жизни объектов данных повышает эффективность работы процедур управления памятью. Например, в языке C все объекты, объявленные в функции, имеют одинаковое время жизни и могут быть расположены в единой области памяти, которая удаляется при выходе из функции.

- *Выбор операции.* В большинстве языков используются специальные символы для обозначения родственных операций. Например, символ «+» обозначает как целочисленное, так и вещественное сложение. Говорят, что такой символ является *перегружаемым*, поскольку может обозначать целое семейство операций сложения, каждая из которых реализует сложение объектов своего типа. Объявления позволяют компилятору определить во время компиляции, какая именно форма операции подразумевается в конкретном случае.
- *Контроль типов.* С точки зрения повышения надежности программ, объявления имеют очень ценное качество: они позволяют проводить статический контроль типов. Статический контроль выполняется в период компиляции, предваряющий период работы программ.

Статический контроль типов

На аппаратном уровне компьютера контроль типов не осуществляется. Например, аппаратная реализация операции целочисленного сложения не способна проверить, являются ли переданные ей два операнда целыми числами; для нее это просто последовательности битов. Таким образом, на уровне аппаратуры обычные компьютеры очень ненадежны и не реагируют на ошибки в типах данных.

Контроль типов проверяет факт получения каждой операцией (в программе) нужного количества операндов правильного типа. Например, перед выполнением оператора присваивания

```
y = x * b + z
```

компилятор должен проверить наличие у каждой операции (умножения, сложения и присваивания) операндов правильного типа. Если операция умножения * определена только для целых или вещественных чисел, а x принадлежит к символьному типу данных, то фиксируется *ошибка в типе операнда*. Контроль типов может осуществляться в период выполнения программы (динамический контроль типов) или в период компиляции (*статический контроль типов*). Значительное преимущество применения универсальных языков программирования состоит в том, что в их реализациях предусмотрен контроль типов для операций. Отсюда вывод — программист защищен от этих коварных ошибок.

Большинство традиционных императивных и объектно-ориентированных языков программирования ориентировано на повсеместное применение статического контроля типов (в период компиляции). Необходимые при этом сведения частично поступают из явных объявлений.

Перечислим исходные данные, необходимые для организации статического контроля типов.

Во-первых, для каждой операции определены количество и типы данных для операндов и результата.

Во-вторых, тип каждого объекта данных (переменной или экземпляра типа) известен и не меняется в ходе выполнения программы.

В-третьих, типы всех констант тоже понятны. Тип литерала легко определяется из синтаксиса его записи в тексте программы, а тип именованной константы зафиксирован в ее определении.

Перечисленная информация собирается компилятором при анализе (просмотре) текста программы и заносится в таблицу символов, которая накапливает все сведения о типах переменных и операций. После завершения сбора компилятор проверяет все операции программы на предмет правильности типов их операндов. После проверки типов операндов *i*-й операции определяется тип ее результата, а полученная информация сохраняется для проверки следующей, (*i*+1)-й операции программы, в которой результат *i*-й операции может использоваться как операнд.

Поскольку статический контроль типов охватывает все операции программы, то проверке подвергаются все варианты вычислений и отпадает необходимость в дальнейшем контроле. В силу этого не требуется включать в контейнеры объектов данных сложные дескрипторы и производить контроль типов на этапе вычислений. В итоге значительно выигрывают в скорости вычислений и эффективности использования памяти.

Специфика статического контроля типов существенно воздействует не только на объявления и структуру текста программы, но и на системные механизмы управления данными в период вычислений, организацию самого процесса компиляции, включая возможности раздельной компиляции программных фрагментов и т. д. В некоторых языках статический контроль типов для определенных конструкций просто невозможен. В подобных ситуациях рассматриваются две возможности:

1. Заменить статический контроль на динамический. Правда, при этом резко возрастают затраты памяти, поскольку в период вычислений придется хранить дескрипторы для всех объектов данных (невзирая на то, что проверяются они достаточно редко).
2. Полностью отказаться от идеи контроля типов. Конечно, это чревато появлением серьезных и трудноуловимых ошибок, но иногда приходится принимать подобное решение, если цена динамического контроля типов слишком высока.

Динамический контроль типов

Динамический контроль типов для операндов некоторой операции осуществляется в период вычислений, непосредственно перед исполнением операции. Реализация динамического контроля базируется на использовании сложного дескриптора, который хранится вместе с объектом данных и описывает его тип и другие атрибуты. Например, в логическом объекте данных находится и его значение, и дескриптор, свидетельствующий о принадлежности объекта к логическому типу. При выполнении любой операции проверяется дескриптор каждого операнда. Операция производится, только если типы операндов правильны; в противном случае формируется сообщение об ошибке. Мало того, к результату операции тоже приписывается дескриптор, характеризующий его тип. Следовательно, будущие операции, которые задействуют данный объект как операнд, тоже смогут проверить его тип.

Динамический контроль типов характерен для скриптовых, функциональных и логических языков. В этих языках отсутствуют явные объявления типов переменных. Типы у переменных могут меняться в ходе выполнения программы. Переменные называются *не имеющими типа*, поскольку в действительности у них нет

никакого постоянного типа. Возможность исполнения любой операции проверяется путем динамического контроля типов операндов.

Основным преимуществом динамического контроля считают гибкость программы. Так как объявлений типов нет, объект данных может поменять свой тип в любой момент вычислений. Программист может создавать настраиваемые программы, способные работать с данными любого типа. Наряду с достоинствами динамический контроль типов вносит ряд существенных недостатков:

1. *Понижение надежности вычислений.* Фактически работа над возможными ошибками типизации переносится с подготовительного этапа, этапа компиляции, на исполнительный этап — этап вычислений. При этом степень охвата ошибок сокращается, минимизируются возможности устранения ошибок. Если при компиляции речь шла о тотальном контроле всех операций программы, то здесь проверке подвергаются лишь отдельные объекты данных, задействованные в текущих вычислениях. Многие из объектов остаются вне контроля. Например, вне поля зрения остаются объекты данных, обрабатываемые в других режимах работы программы, которые понадобятся в обозримом будущем.
2. *Снижение скорости вычислений.* Динамический контроль типов основан на информации, которая хранится в дескрипторах объектов данных. Как правило, доступ к дескрипторам должен быть реализован на программном уровне, так как аппаратура таких действий не обеспечивает. Это выливается в существенное замедление скорости вычислений.
3. *Возрастание накладных затрат памяти.* Поскольку каждый объект данных несет в себе сложный дескриптор, который содержит обширные сведения и должен храниться в течение всего вычислительного процесса, динамический контроль типов предъявляет повышенные запросы к требуемой памяти.

Обзор составных типов данных

Объекты данных таких типов характеризуются наличием многочисленных атрибутов.

Проиллюстрируем самый популярный составной тип данных — массив (рис. 8.1).

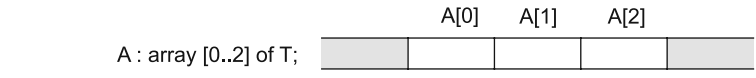


Рис. 8.1. Объект данных типа «массив»

На рис. 8.1 показана лента памяти, в которой размещается экземпляр типа «массив». Все элементы массива примыкают друг к другу, образуя в памяти непрерывную ленту, имеют один тип и занимают одинаковую порцию памяти. Массив поддерживает произвольный способ доступа к его элементам. Имя *i*-го элемента массива `A` записывается как `A[`*i*`]`, где *i* называется индексом элемента. Произвольный доступ означает, что время доступа к `A[`*i*`]` не зависит от значения *i*.

Тип данных «запись» имеет свои достоинства, позволяя внутри одной капсулы разместить разнотипные элементы (рис. 8.2).

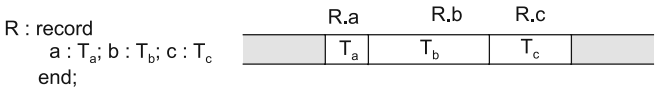


Рис. 8.2. Объект данных типа «запись»

Каждый элемент записи может иметь свой собственный тип и свои правила размещения. Требования «непрерывности» занимаемой «ленты» памяти здесь отсутствуют. Напротив, элементы записи могут быть разбросаны по различным местам памяти компьютера. Элементы (компоненты) записи именуются и называются полями. Запись поддерживает именной доступ к ее полям. Именной доступ означает, что если имя `R` обозначает запись с полем, именуемым как `a`, то поле указывается как `R.a`.

Тип данных «указатель» обозревается в этом разделе, поскольку обычно он обслуживает составные типы данных (рис. 8.3).

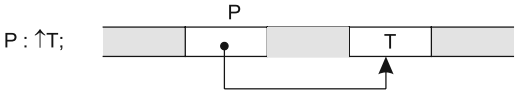


Рис. 8.3. Объект данных типа «указатель»

Указатель, как правило, размещается в машинном слове, независимо от размера объекта данных, на который он указывает. Этому есть простое объяснение: указатель хранит лишь адрес объекта (в данном случае объекта типа `T`).

Наконец, для типа данных «множество» характерна самая высокая плотность хранения данных (рис. 8.4).

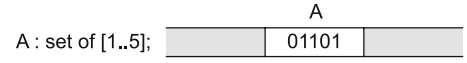


Рис. 8.4. Объект данных типа «множество»

Под элемент множества отводится всего один бит. Этот бит имеет единичное значение, если элемент действительно входит в объект типа множества. В противном случае значение бита равно нулю.

Системы типизации данных

Иногда всю сумму правил, связанных с типами данных и их экземплярами, объединяют в понятие *система типизации данных*. Таким способом стремятся подчеркнуть высокий удельный вес этих понятий и правил, их значимость в теории и практике языков программирования. Системы типизации данных определяют полный жизненный цикл объектов данных в программных вычислениях и включают в себя несколько категорий правил:

1) определения атрибутов переменных, их связывания;
2) определения типов данных;
3) определения типов выражений;
4) правила преобразования и проверки типов данных.

Собственно говоря, все предыдущие разделы были введением в систему типизации данных. Первую и третью категории правил мы рассмотрим в последующих разделах текущей главы, развернутому обсуждению второй категории правил посвящены главы 9, 10 и 11, а правила преобразования и проверки типов данных подробно описываются в главе 12.

Атрибуты переменной

Вернемся еще раз к обсуждению фундаментального понятия языков программирования — понятию переменной. С одной стороны, с точки зрения аппаратуры, переменная является абстракцией ячейки памяти компьютера. С другой стороны, с точки зрения программы, переменная — это абстракция объекта данных.

Несмотря на кажущуюся простоту, переменная является достаточно сложным объектом, имеющим много свойств. Свойства переменной можно пояснить с помощью шести атрибутов (характеристик), в состав которых входят: имя, тип, адрес, значение, время жизни, область видимости. Первые пять атрибутов рассмотрим сейчас (рис. 8.5). Область видимости — это сегмент, фрагмент программы, в пределах которой переменная доступна. Ее мы обсудим чуть позже, в главах 13 и 14.

Рис. 8.5. Основные атрибуты переменной

Имя идентифицирует переменную в программе, является самой распространенной (но не единственной) категорией программных имен, которые подробно обсуждались в главе 6. Синонимом слова «имя» является слово «идентификатор». Большинство переменных имеют имена, хотя и не все. Безымянные переменные мы будем обсуждать особо.

Тип определяет возможные значения, операции, способ размещения переменной в памяти. Например, для типа `int` в языке Java определен диапазон значений от –2147483648 до +2147483647 и разрешены операции сложения, вычитания, умножения, деления и деления по модулю.

Адрес указывает ячейку памяти под переменную. Поскольку в компьютерах ячейки имеют разные размеры, удобен термин «абстрактная ячейка».

Адрес переменной является адресом той области машинной памяти, где она хранится. Соответствие между переменной и адресом не такое простое, как может показаться на первый взгляд. Во многих языках одна и та же переменная имеет разные адреса в разное время программных вычислений. Например, если в подпрограмме используется локальная переменная, которая берется из стека периода выполнения при вызове подпрограммы, то при разных вызовах подпрограммы у данной переменной будут разные адреса. Это в некотором смысле различные

экземпляры одной и той же переменной. Более подробное объяснение приведено в главе 14.

Адрес переменной иногда называют *левым значением* (*l*-значением), так как имя переменной в левой части оператора присваивания обозначает именно адрес.

Достаточно часто несколько переменных имеют один и тот же адрес. Если для доступа к одной и той же области памяти используют несколько имен переменной, такие переменные называют *алиасами*.

Применение алиасов затрудняет чтение программ, поскольку разрешает изменять значение одной и той же области памяти при помощи нескольких переменных. Например, если переменные с именами `total` и `sum` являются алиасами, любое изменение значения `total` изменяет значение `sum`, и наоборот. Человек, читающий программу, всегда должен помнить, что `total` и `sum` — это разные имена одной и той же ячейки памяти. Поскольку в программе может быть любое количество алиасов, запомнить все это достаточно сложно. Кроме того, использование алиасов усложняет проверку программы.

В программах алиасы могут создаваться различными способами. Например, могут применяться такие типы данных, как вариантные записи и объединения (они рассматриваются в главе 10).

Две переменные-указатели считаются алиасами, если они указывают на одну и ту же область памяти. Это справедливо и для переменных-ссылок. Такая разновидность алиасов является побочным эффектом природы указателей и ссылок. Если указатель в языке C++ адресует переменную с именем, то разыменованный указатель и переменная с именем становятся алиасами.

Во многих языках алиасы создаются с помощью параметров подпрограмм. Эта разновидность алиасов обсуждается в главе 13.

Для понимания языков программирования очень важен момент связывания переменной со своим адресом. Подробнее этот вопрос рассмотрим чуть позже.

Значение переменной характеризуется содержимым ячейки (области) памяти под переменную. Память компьютера удобно представлять в терминах не физических, а абстрактных ячеек. Ячейки большинства компьютеров, являющиеся прямо адресуемыми единицами, имеют размер, равный 8-разрядному байту. Этот размер слишком мал для большинства программных переменных. Будем полагать, что абстрактная ячейка памяти имеет размер, достаточный для хранения соответствующей переменной. Например, хотя в реализации числа с плавающей точкой занимают четыре физических байта, вообразим, что число с плавающей точкой размещается в одной абстрактной ячейке памяти. Такое же допущение принимаем для экземпляра любого элементарного типа данных.

Значение переменной иногда называют *правым значением* (*r*-значением), поскольку именно его обозначает имя переменной, указанное в правой части оператора присваивания. Для получения доступа к правому значению переменной надо знать ее левое значение. Значительно усложнить этот процесс могут, например, правила видимости, обсуждаемые в главах 13 и 14.

Время жизни переменной определяется периодом существования переменной в программе. На это время переменной назначается ячейка памяти, хранящая ее значение. Более подробное объяснение основывается на концепции связывания, к рассмотрению которой мы и переходим.

Связывание

В жизненном цикле переменной ее атрибуты необходимо связывать. *Связывание* (binding) — это процесс установления связи между атрибутами. Момент времени, когда эта связь устанавливается, называют *временем связывания*.

Связывание и время связывания — важнейшие понятия как семантики всего языка программирования, так и его системы типизации данных. Связывание может происходить во время проектирования языка, при разработке компилятора языка, при компиляции, загрузке или выполнении программы. Символ плюс (+), например, обычно связывается с операцией сложения во время проектирования языка. Тип данных, например `int` в языке C, связывается с диапазоном возможных значений во время разработки компилятора. В языке Java переменная связывается с конкретным типом данных во время компиляции программы. Переменная может связываться с ячейкой памяти при загрузке программы в память. Аналогичное связывание иногда задерживается до начала выполнения программы, например, для переменных, объявленных в методах языка Java. Вызов библиотечной подпрограммы связывается с кодом подпрограммы на этапе редактирования связей и загрузки.

Как правило, программист-практик выносит за скобки этапы создания языка и реализующего его компилятора. Он ограничивается лишь этапами компиляции и выполнения программы. Условимся, что в дальнейшем мы будем обсуждать только эти два этапа.

Приведем два примера связывания.

Рис. 8.6. Связывание типа

Связывание типа — имя связывается с типом (рис. 8.6):

```
var  x : integer;      /* Pascal */
int  y;                /*    C   */
```

Рис. 8.7. Связывание значения

Связывание значения — имя связывается со значением (рис. 8.7):

```
x := 15;     /* Pascal */
y = 25;      /*    C   */
```

Рассмотрим оператор присваивания для языка Java:

```
sum = sum + 7;
```

Перечислим необходимые связывания для компонентов этого оператора:

- Тип переменной `sum` — связывание во время компиляции.
- Смысл символа операции `+` — связывание во время компиляции, после определения типов операндов.
- Внутреннее представление константы `7` — связывание во время компиляции.

- Значение переменной `sum` — связывание во время выполнения указанного оператора.

Понимание связывания атрибутов с сущностями программы является необходимым условием для освоения семантики языка. В частности, следует разобраться, как фактические параметры оператора вызова связываются с формальными параметрами подпрограммы. Для определения текущего значения переменной нужно понимать механизм связи переменной с ячейкой памяти и необходимыми операциями.

По времени различают две разновидности связывания: статическое и динамическое (рис. 8.8):

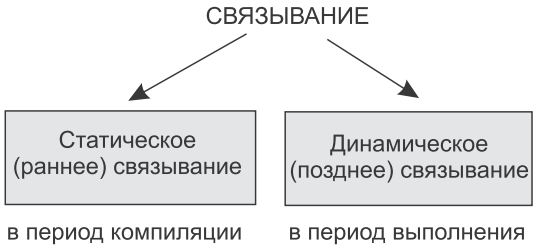


Рис. 8.8. Разновидности связывания

Статическое связывание выполняется в период компиляции, а динамическое — в период выполнения программы.

На аппаратном уровне и уровне операционной системы процесс связывания достаточно сложен, так как адресуемые ячейки памяти подвергаются многократной загрузке/выгрузке в виртуальном пространстве компьютера. Оставляя эти подробности без внимания, сосредоточимся на специфике статического и динамического связывания.

По сути, статическое связывание типа мы уже обсудили при рассмотрении явных и неявных объявлений. Эти объявления являются единственными инструментами связывания имен переменных с типами на этапе компиляции. Перейдем к рассмотрению динамического связывания типов.

Динамическое связывание типов

При динамическом связывании оператор объявления переменной не содержит имени типа. Вместо этого тип переменной определяется при присвоении ей значения оператором присваивания. При выполнении оператора присваивания переменная из его левой части получает тип переменной, выражения или значения, находящегося в его правой части.

Такое присваивание может связать переменную с новым адресом и новой ячейкой памяти, потому что форматы хранения величин в разных типах отличаются друг от друга. Динамическое связывание постулирует: любой переменной может быть присвоено значение любого типа. Более того, во время выполнения программы тип переменной может меняться многократно. Важно понимать, что тип у переменной с динамически связанным типом имеет лишь временный характер.

Если тип связан с переменной статически, то имя переменной навсегда привязано к этому типу. Напротив, если тип связан с переменной динамически, то имя переменной лишь временно привязано к типу. На самом деле имена таких переменных никогда не ограничены типами. Более точно: имена могут быть связаны с переменными, а переменные могут быть связаны с типами.

Языки с динамической типизацией существенно отличаются от языков со статическим связыванием типов. Основное преимущество динамического связывания переменных с типом заключается в повышении гибкости программирования. В частности, программу обработки числовых данных в языке, применяющем динамическое связывание типов, можно написать в форме настраиваемой программы. Такая программа сможет работать с данными любого числового типа. Любой тип входных данных считается приемлемым, поскольку переменные, предназначенные для их хранения, будут привязываться к этому типу в ходе присваивания. Напротив, статическое связывание типов не позволяет написать на языке C программу для обработки данных без фиксации их типа.

До середины 1990-х годов самые популярные языки программирования повсеместно использовали статическое связывание типа, редким исключением были некоторые функциональные языки, такие как LISP. Однако с тех пор произошел существенный сдвиг в сторону динамического связывания типов. В языках Python, Ruby, JavaScript и PHP теперь принято динамическое связывание типа. Например, программа на языке JavaScript может содержать следующий оператор:

```
list = [4.3, 8.2];
```

Независимо от предыдущего типа переменной `list`, в результате такого присваивания она превратится в одномерный массив из двух элементов, являющихся числами с плавающей точкой. Если же далее последует оператор

```
list = 75;
```

то переменная `list` станет целочисленной скалярной переменной.

Возможность динамического связывания типа была введена в язык C# версии 2010 года. Включение в объявление переменной зарезервированного слова `dynamic` разрешает динамическое связывание типа, это показано в следующем примере:

```
dynamic any;
```

Можно отметить некоторое сходство данного объявления с объявлением

```
object any;
```

Объявления схожи в том, что переменной `any` может быть присвоено значение любого типа (даже типа `object`), как и для объявления с фиксированным типом `object`. Различие же заключается в исключении возможностей взаимодействия с фрагментами на таких динамически типизированных языках, как `IronPython` и `IronRuby` (.NET версий языков Python и Ruby соответственно). Тем не менее это полезно, если данные неизвестного типа поступили в программу из внешнего источника. Члены класса, свойства, параметры метода, возвращаемые методом величины и локальные переменные — все они могут быть объявлены динамичными.

В чисто объектно-ориентированных языках — например, Ruby — все переменные являются ссылками и не имеют типов, а все данные считаются объектами, и любая

переменная может ссылаться на любой объект. В некотором смысле, все переменные в таких языках имеют одинаковый тип — это ссылки. Однако в отличие от ссылок в языке Java, которые ограничены возможностью ссылаться лишь на один конкретный тип, переменные в Ruby могут ссылаться на любой объект.

В заключение еще раз обсудим недостатки динамического связывания типов. Прежде всего снижается надежность программ, поскольку возможности обнаружения ошибок много меньшие, чем у компилятора для языка со статическим связыванием типов. Динамическое связывание типов позволяет присвоить любой переменной значение любого типа. В этом случае неправильные типы на правой стороне оператора присваивания не будут распознаны как ошибки, вместо этого тип переменной-приемника изменится на этот неправильный тип. Допустим, что в конкретной программе на JavaScript имеются две скалярные числовые переменные `a` и `b`, а также массив `m`. Допустим также, что в программе должен быть оператор присваивания:

```
a = b;
```

Вместо него был введен неправильный оператор:

```
a = m;
```

В языке JavaScript (или любом другом языке с динамическим связыванием типов) интерпретатор ошибку не обнаружит. Тип переменной `a` просто будет изменен на тип массива. Поскольку при дальнейшем использовании переменной `a` ожидалось скалярное значение, результаты станут непредсказуемыми. В языке со статическим связыванием типов, таком как Java, компилятор обнаружит ошибку `a = m`, и программа не будет выполнена.

Отметим, что этот недостаток частично присутствует и в языках со статическим связыванием типов (Fortran, C и C++), которые в некоторых случаях автоматически приводят результат из правой части оператора присваивания к типу переменной из его левой части.

Самым большим недостатком динамического связывания типов является его стоимость. Стоимость реализации динамического связывания атрибутов весьма значительна, особенно во время вычислений. Именно в это время должна выполняться проверка типов. Для ее обеспечения каждая переменная несет в себе сложный дескриптор. Для хранения любой переменной задействуется область памяти переменного размера, поскольку формат сохраняемого значения меняется от типа к типу.

В языках с динамическим связыванием типов чаще всего используются интерпретаторы. У аппаратных средств компьютера нет команд, типы операндов которых неизвестны в период компиляции. Потому компилятор не может создавать машинные команды для выражения `x + y`, где типы `x` и `y` неизвестны в период компиляции. Чистая интерпретация, как правило, выполняется в 10 раз медленнее, чем эквивалентный машинный код. Конечно, если язык реализуется чистым интерпретатором, то время на выполнение динамического связывания типов скрыто и кажется более дешевым. С другой стороны, в языках со статическим связыванием типов обычно применяются компиляторы, так как программы на этих языках транслируются в очень эффективный машинный код.

Время жизни

Жизнь переменных напрямик связана с их ресурсным обеспечением. Конечно, главным требуемым ресурсом является память. Каждая переменная должна «владеть» своей порцией памяти. Динамически распределяемый ресурс памяти принято называть *пулом* (pool). Обычно говорят о пуле свободной памяти, откуда берут порцию памяти. Размер этого пула в ходе вычислений часто изменяется.

Жизнь переменной начинается с ее *размещения в памяти* (allocation). При размещении происходит следующее:

1) из пула свободной памяти извлекается ячейка памяти;
2) извлеченная ячейка связывается с переменной.

Жизнь переменной прекращается при ее *удалении из памяти* (deallocation):

1) связь переменной с ячейкой памяти разрывается (переменная «открепляется» от ячейки);
2) ячейка памяти возвращается в пул свободной памяти.

Время жизни переменной — это период времени, в течение которого переменная связана с определенной ячейкой памяти. Таким образом, время жизни переменной начинается при ее связывании с определенной ячейкой памяти и заканчивается при ее откреплении от этой ячейки. Для обсуждения связывания переменных с ячейками памяти удобно выделить четыре разновидности переменных (согласно времени их жизни). Назовем данные разновидности статическими, стековыми, явными динамическими и неявными динамическими переменными. В следующих подразделах рассматриваются цели, достоинства и недостатки этих четырех разновидностей.

Статические переменные

Статической называют переменную, которая связывается с ячейкой памяти до начала выполнения программы и сохраняет связь с той же самой ячейкой памяти вплоть до завершения программы.

Статические переменные очень полезны в программировании. Достаточно часто в программах применяют *глобальные переменные*, которые должны быть доступны во всех местах программы. Кроме того, глобальные переменные обычно нужны в течение всего времени выполнения программы. Сумма этих свойств свидетельствует о целесообразности статического связывания глобальных переменных с памятью. Иногда требуется, чтобы переменные, объявляемые в подпрограммах, сохраняли свои значения между отдельными выполнениями подпрограммы. Это означает, что такие подпрограммы должны иметь *локальные статические* переменные.

Другим достоинством статических переменных считается их эффективность. Дело в том, что при обращении к статическим переменным используется быстрая прямая адресация, тогда как другие разновидности переменных зачастую нуждаются в более медленной косвенной адресации. По этой же причине минимизируются затраты времени на размещение и удаление статических переменных.

К недостатку статического связывания с памятью относят уменьшение гибкости. Например, в языках, которые поддерживают только статические переменные, нельзя применять рекурсивные подпрограммы. Дополнительным минусом

считается невозможность использования несколькими переменными одной и той же области памяти. Допустим, что в программе есть две подпрограммы, причем в каждой объявляется свой собственный статический массив. Несмотря на то что подпрограммы никогда не вызываются одновременно, память под массивы нельзя использовать совместно.

В языках C и C++ разрешается спецификатор `static` в объявлении локальных переменных для функции; соответствующие переменные при этом считаются статическими. Применение `static` для объявления переменной в классе C++, Java и C# означает, что переменная становится переменной класса, а не переменной экземпляра (объекта). Переменные класса создаются статически за некоторое время до начальной инициализации класса.

Стековые переменные

Стековыми называются переменные, удовлетворяющие двум условиям:

- связывание с памятью осуществляется при *обработке* операторов объявления переменных;
- типы переменных связываются статически.

Обработка (elaboration) такого *объявления* приводит к распределению памяти и реализации процессов связывания, указанных в объявлении. Эти действия происходят при достижении фрагмента программного кода, с которым связано объявление, в процессе *выполнения* программы. Важно подчеркнуть, что обработка происходит во время вычислений.

Например, объявления переменных, которые записаны в начале Java-метода, обрабатываются при вызове метода, а созданные по этим объявлениям переменные будут удалены, когда метод завершит свою работу. Как следует из их названия, память стековым переменным выделяется из стека периода прогона (выполнения).

Некоторые языки — например, C++ и Java — позволяют размещать объявления переменных в любом месте операторов. В реализациях этих языков все стековые переменные, объявленные в функции или методе (не считая те переменные, которые объявлены во вложенных блоках), связываются с памятью только с началом выполнения функции или метода. В таких случаях переменная становится видимой с момента объявления, но связывание с памятью (и инициализация, если она указана в объявлении) происходит лишь тогда, когда функция или метод начинает выполняться. Тот факт, что видимость переменной опережает ее связывание с памятью, не влияет на семантику языка.

Обсудим достоинства стековых переменных. В большинстве случаев рекурсивным подпрограммам требуется такая форма динамической локальной памяти, которая обеспечивает для каждой активной копии рекурсивной подпрограммы свою собственную версию локальных переменных. Эта потребность удовлетворяется стековыми переменными. Наличие у подпрограмм стековой локальной памяти полезно даже при отсутствии рекурсии, поскольку все подпрограммы могут совместно использовать одну и ту же область памяти для хранения своих локальных переменных.

Недостатки стековых переменных (по сравнению со статическими переменными):

- дополнительные затраты времени на размещение в памяти и удаление из памяти;
- уменьшение скорости доступа из-за использования косвенной адресации;
- подпрограммы не имеют возможности хранить предысторию вычислений.

Впрочем, время на размещение/удаление стековых переменных минимизируется за счет того, что переменные объявляются в начале подпрограммы и размещаются/удаляются все вместе, а не отдельно каждая переменная.

Язык Fortran 95 позволяет разработчикам использовать стековые переменные для локальных вычислений, но содержит оператор

```
Save list
```

Этот оператор вставляется в подпрограмму и объявляет в качестве статических все переменные, перечисленные в списке `list`.

В языках Java, C++ и C# локальные переменные в методах становятся стековыми переменными по умолчанию. В языке Ada все нединамические переменные, определенные в подпрограммах, считаются стековыми.

Все атрибуты и характеристики переменной, кроме памяти, статически связываются со стековыми переменными. Исключения для некоторых составных типов обсуждаются в главе 10. Реализация процессов размещения в памяти и удаления из нее стековых переменных описана в главе 14.

Явные динамические переменные

Явные динамические переменные — это безымянные ячейки памяти, размещаемые и удаляемые с помощью явных операторов программы, которые задействуются в период выполнения. Обращаться к этим переменным, располагаемым в области памяти по имени куча, можно только с помощью указателей и ссылок. *Куча* (heap) состоит из коллекции ячеек памяти с весьма неорганизованной структурой, вызванной непредсказуемостью их применения. Явные динамические переменные создаются или оператором (например, в языке C++), или вызовом предусмотренной библиотечной подпрограммы (например, в языке C).

В языке C++ существует оператор размещения `new`, операндом которого является имя типа. При выполнении этого оператора в куче создается явная динамическая переменная, имеющая тип операнда, и возвращается указатель на нее. При создании явной динамической переменной задействуется как статическое, так и динамическое связывание. Поскольку явная динамическая переменная связывается с типом во время компиляции, то связывание типа здесь считается статическим. С другой стороны, подобные переменные связываются с ячейками памяти в момент их создания, то есть динамически, при выполнении программы.

Помимо операторов либо подпрограмм для размещения явных динамических переменных, в некоторых языках есть средства их удаления.

Проиллюстрируем работу с явной динамической переменной с помощью средств языка C++:

```
int *pointer;        // создать указатель
pointer = new int;   // разместить явную динамическую переменную
delete pointer;      // удалить явную динамическую переменную,
                     // на которую указывает указатель pointer
```

В этом фрагменте создается указатель `pointer`, способный хранить адрес явной динамической переменной с типом `int`. Далее в куче с помощью оператора `new` размещается явная динамическая переменная, имеющая тип `int`. Адрес этой безымянной переменной заносится в указатель `pointer`. Затем переменная удаляется из кучи оператором `delete`. Язык C++ требует явного удаления с помощью оператора `delete`, поскольку здесь не используется неявная очистка памяти, такая как сборка мусора.

В языке Java все данные, за исключением основных скалярных величин, являются объектами. Объекты языка Java представляют собой явные динамические объекты, доступные через ссылочные переменные. В языке Java нет средств явного удаления динамических переменных; вместо этого используется неявная сборка мусора.

В языке C# имеются как явные динамические, так и стековые объекты, все они удаляются неявно. Кроме того, C# поддерживает стиль указателей языка C++. Такие указатели используются для ссылок на кучу, стек и даже на статические переменные и объекты. Эти указатели несут ту же степень опасности, что и в C++, а объекты, на которые они ссылаются в куче, не могут быть удалены неявно. Указатели включены в C# для поддержки взаимодействия с компонентами C и C++. Чтобы воспрепятствовать их применению и довести до читателя тот факт, что в коде используются указатели, заголовок любого метода, где определяется указатель, должен содержать зарезервированное слово `unsafe` (небезопасно).

Явные динамические переменные востребованы в динамических структурах (например, связанных списках, деревьях), размеры которых динамически изменяются в период вычислений. Подобные структуры проще всего создавать на основе указателей или ссылок, а также явных динамических переменных.

Недостатком явных динамических переменных являются трудности корректного использования указателей и ссылок, стоимость ссылок на переменные, а также сложность реализации требуемого механизма управления памятью. Особенно существенна проблема сложности и стоимости управления кучей. Все эти соображения и вопросы реализации явных динамических переменных подробно обсуждаются в главах 11 и 14.

Неявные динамические переменные

Неявные динамические переменные связываются с динамической памятью «куча» только при присваивании им значений. Фактически все их атрибуты и характеристики вновь связываются при каждом присвоении переменной нового значения. Например, рассмотрим оператор присваивания на языке JavaScript:

```
volumes = [24, 94, 76, 50, 41];
```

Независимо от того, использовалась ли переменная `volumes` раньше и для чего она использовалась, теперь это массив из пяти числовых значений.

Достоинством таких переменных считается высокая степень гибкости, позволяющая писать обобщенные программы.

Одним из недостатков неявных динамических переменных являются высокие накладные расходы на обслуживание всех динамических атрибутов, среди которых могут быть: тип элементов массива, диапазон индексов и многие другие. К другому недостатку следует отнести потерю компилятором возможностей распознавания многих ошибок.

Тип выражения

Как и каждая переменная, каждое выражение тоже должно иметь тип. Он должен быть известен и зафиксирован в период выполнения. Этот тип зависит от типа переменных в выражении, а также от операций выражения. Например, выражение 2 + 3 имеет целый тип, так как и 2 и 3 являются целыми числами, а сумма целых — тоже целое число.

Тип выражения логически выводится в период компиляции.

Набор правил для определения типа выражения входит в *систему типизации языка*. Система типизации отвергает выражение, если оно не сопоставимо с типом.

Правила системы типизации определяют корректность использования каждой операции языка в выражении.

В качестве простейшего, исторического примера типизации выражений опишем систему для языка Фортран.

Пример. Система типизации для арифметических выражений в начальном Фортране

Выражение является или переменной, или константой, или формируется применением одной из операций `+`, `-`, `/`, `*` к двум подвыражениям. Тип выражения: `int` или `real`.

Выражение имеет тип, если, и только если, к нему применимо одно из правил:

- Переменные с именами, которые начинаются с букв `I . . N`, имеют тип `int`, переменные с другими именами имеют тип `real`. Например: `COUNT` имеет тип `real`.
- Число имеет тип `real`, если содержит десятичную точку, в противном случае — `int`. Например: 0.5, .5 , 5. , 5.0 — все имеют тип `real`.
- Классификация переменных и констант на `int` и `real` распространяется и на выражения. Если выражения `E` и `F` имеют одинаковый тип, то `E + F`, `E – F`, `E / F`, `E * F` — выражения того же типа.
- Например, выражение `I + J` имеет тип `int`, выражение `X + Y` имеет тип `real`. Выражение `I + X` в стандартном Фортране не разрешено.

Арифметические операции

С каждой операцией (`op`) сопоставляется правило, которое определяет тип выражения `E op F` в терминах типов для `E` и для `F`. Например: если `E` и `F` имеют тип `int`, то и `E + F` имеет тип `int`.

Перегрузка операции — множественность смыслов

Некоторые символы операций (`+`, `*`) являются перегружаемыми, то есть в разных случаях они имеют разный смысл.

В Фортране `+` обозначает как целое, так и вещественное сложение. Поэтому выражение сложения имеет два возможных типа: `real`, `int`.

Следовательно, для сложения нужны два правила:

1. Если `E` и `F` имеют тип `int`, то `E + F` также имеет тип `int`;
2. Если `E` и `F` имеют тип `real`, то `E + F` также имеет тип `real`.

Приведение — преобразование типов

Система типизации в оригинальном Фортране отвергала выражения `X + I` и `2 * 3.142`, так как один операнд в них был целым, а другой — вещественным числом.

Конечно, с современной точки зрения все это выглядит очень наивно. Современные системы типизации выражений демонстрируют совсем другой уровень гибкости — пройден достаточно длинный путь эволюции. В частности, большинство современных языков программирования рассматривают выражение 2 * 3.142 как 2.0 * 3.142, то есть как произведение двух вещественных чисел за счет приведения. Приведение — это преобразование объекта данных из одного типа в другой, выполняемое неявно (автоматически) или явно.

Контрольные вопросы и упражнения

1. Предложить пользователю ввести целое число. Закончить приглашение двоеточием, за которым следует пробел, установить курсор после пробела.
2. Сосчитать с клавиатуры целое число и запомнить введенное значение в целой переменной `a`.
3. Напечатать сообщение «Это программа на Си» на одной строке.
4. Напечатать сообщение «Это программа на Си» на двух строках, где первая строка заканчивается на слове «программа».
5. Написать сообщение «Это программа на Си» так, чтобы на каждой строке было только одно слово.
6. Напечатать сообщение «Это программа на Си» так, чтобы каждое слово было отделено от следующего знаком табуляции.
7. Написать текст, соответствующий следующему:
   а) заявить, что программа будет вычислять произведение трех целых чисел;
   б) объявить переменные `x`, `y`, `z`, `rezult` целого типа;
   в) предложить пользователю ввести три целых числа;
   г) сосчитать три целых числа с клавиатуры и сохранить их в переменных `x`, `y`, `z`;
   д) вычислить произведение трех целых чисел, содержащихся в переменных `x`, `y`, `z` и присвоить результат переменной `result`;
   е) на экране напечатать «произведение равно», а потом значение переменной `result`;
   ж) используя операторы из предыдущих заданий, оформить полную программу.

Операции присваивания

В C имеется несколько операций присваивания, которые позволяют сократить запись присваиваемых выражений.

Например, оператор `c = c + 3` можно записать как `c += 3`.

```
e = e * 5  ⇔ e *= 5
d = d – 4  ⇔ d -= 4
g = g % 9 ⇔ g %= 9
```

Если операции инкремента и декремента записаны в префиксной форме, то переменная сначала изменяется, а затем ее новое значение используется в выражении.

Если операции инкремента и декремента записаны в постфиксной форме, то в выражении используется текущее значение переменной, и только после этого ее значение изменяется.

Замечание: если операции инкремента и декремента осуществляются в виде отдельного оператора, то обе формы приводят к одинаковому результату.

8. Каким будет значение переменной `c` в каждой строке?

```
#include <stdafx.h>
#include <stdio.h>
int main()
{ int c = 5;
    printf("%d\n",c);     //
    printf("%d\n",c++);   //
    printf("%d\n",c);     //
    c = 5;
    printf("%d\n",c);     //
    printf("%d\n",++c);   //
    printf("%d\n",c);     //
    return 0;
}
```

9. Выберите элементарный тип данных, определенный в хорошо известном вам языке программирования, и проделайте следующее:
   - Опишите множество значений, которые могут содержаться в объектах данных этого типа.
   - Определите, как представлены в памяти значения этого типа данных (в конкретной реализации языка, используемой на вашем компьютере).
   - Определите синтаксическое представление для констант этого типа данных.
   - Определите набор операций, определенных для объектов данных выбранного вами типа. Для каждой такой операции приведите ее сигнатуру и синтаксическое представление в языке.
   - Для каждой операции определите, как она реализована: при помощи программного моделирования или непосредственно как одна команда процессора.
   - Опишите какие-нибудь атрибуты объектов данных выбранного типа, отличные от атрибутов этого типа данных.
   - Определите, являются ли какие-нибудь символы или имена операций, определенных для этого типа данных, перегружаемыми. Для каждого перегружаемого имени или символа операции определите, когда (при компиляции или при выполнении программы) задается специфический смысл каждого использования перегружаемого имени в операторе.
   - Определите, статический или динамический контроль типов применяется для проверки правильности каждого использования каждой операции, определенной для выбранного типа данных.
10. Выберите хорошо известный вам язык программирования и для какого-либо определенного в нем элементарного типа данных сделайте следующее:
    - Объясните разницу между типом, переменными этого типа и константами этого типа.
    - Приведите пример ситуации, возникающей в процессе выполнения программы, когда объект данных выбранного вами типа существует, но не является ни переменной, ни константой.

- Объясните разницу между объектами данных этого типа и значениями, которые они могут содержать.

11. Выберите хорошо известный вам язык программирования и приведите пример элементарной операции, которая
    - имеет неявный аргумент;
    - приводит к побочным эффектам;
    - не определена для некоторых объектов данных из своей области определения;
    - зависит от предыстории.
12. Приведите примеры таких конструкций из языка программирования со статическим контролем типов, которые не могут быть статически проверены. Для каждой конструкции напишите тестовую программу, которая позволит выяснить, проверяется ли эта конструкция динамически или так и остается не проверенной во время выполнения программы.

Глава 9
Скалярные типы данных

В данной главе рассматривается семейство элементарных типов данных под общим названием скалярные типы: перечисления, целые и вещественные типы, десятичные числа. Вводится понятие поддиапазона, который ограничивает диапазон значений существующего базового типа, обсуждается логический тип и символьные типы. В завершение комментируется стиль программирования на языке C, основанный на неявном преобразовании типов.

Дескриптор каждого объекта скалярного типа хранит лишь пару-тройку атрибутов. Например, для целочисленного объекта единственными атрибутами являются его имя и значение, а другой информации в этом объекте нет.

Вообще специфика объектов данных скалярных типов во многом определяется возможностями аппаратуры компьютера.

Перечисления

Перечисление — это тип данных, представляемый конечной последовательностью имен, записанных внутри скобок.

Объявление

```
type день = (пон, втр, срд, чтв, птн, сбт, вск);
```

вводит тип `день` как перечисление из семи элементов. В языке Pascal любое имя из перечисления (например, `пон`) считается константой (значением). Для каждого вхождения константы распознается ее принадлежность к типу `день`.

Скалярный тип `boolean` рассматривается как предопределенное перечисление

```
type boolean = (false, true);
```

Аналогично, тип `char` является перечислением, которое определяется набором машинных символов. Широко используемый набор *ASCII* (American Standard Code for Information Interchange) включает 128 символов.

Символьная константа заключается в одиночные кавычки (апострофы) `'&'`. Одиночные кавычки обозначаются `''`, поэтому символьная константа для одиночных кавычек записывается так `''''`.

Элементы перечисления упорядочены, это значит, что

```
пон < втр < .. < вск
```

К элементам перечисления (и к целым числам) применимы следующие операции языка Pascal:

- `ord (x)` — функция определяет для имени `x` его номер в последовательности, поэтому `ord (вск) = 7`;
- `succ (x)` — функция (successor) определяет для имени `x` следующее имя в последовательности, поэтому `succ (втр) = срд`; если `x` — последнее имя, то фиксируется ошибка;
- `pred (x)` — функция (predecessor) определяет для имени `x` предшествующее имя в последовательности, поэтому `pred (втр) = пон`; если `x` — первое имя, то фиксируется ошибка.

Например, для дней может быть реализована функция `завтра (x)`:

```
if x = вск then завтра := пон else завтра := succ (x)
```

Перечисления в языке Ada подобны перечислениям языка Pascal, за исключением того, что имена-значения могут появляться в объявлениях нескольких типов. Такие имена называются перегружаемыми [52].

В языке C перечисление объявляется следующим образом [14]:

```
enum day {пон, втр, срд, чтв, птн};
```

В отличие от языка Pascal, элементы перечисления здесь считаются именами целых констант (принадлежащих к целому типу). Иначе говоря, в примере элементам в фигурных скобках автоматически назначаются числовые значения от 0 до 4:

```
пон = 0, втр = 1, срд = 2, чтв = 3, птн = 4
```

В языке C++ элементы перечисления тоже имеют порядковые номера, но эти номера (числа) не могут быть использованы как их эквиваленты (значения). Например, после объявления

```
day today;
```

присваивание

```
today = 4;
```

считается некорректным. Оно может быть разрешено только в том случае, если число из правой части будет *явно* преобразовано в тип `day`. Такой запрет предотвращает некоторые потенциальные ошибки.

В языке C# перечисления подобны C++, за исключением того, что их элементы никогда не преобразуются в целые числа [38]. Таким образом, здесь существенно ограничены как значения, так и операции над перечислениями.

В 2004 году перечисления были добавлены в язык Java версии 5.0 [44]. Все перечисляемые типы здесь считаются неявными подклассами предопределенного класса `Enum`. Поскольку перечисляемые типы являются классами, они могут иметь поля данных, конструкторы и методы. Синтаксически определения перечисляемого типа в Java подобны C++ (если не учитывать поля, конструкторы и методы). Экземпляры перечислений являются экземплярами классов. Все перечисления наследуют `toString`, а также другие методы. Массив экземпляров перечисляемого типа может быть доступен с помощью статического метода, а внутреннее численное значение перечисляемой переменной — с помощью обычного метода. Перечисляемой переменной нельзя присвоить значение какого-либо другого типа. Кроме того, перечисляемая переменная никогда не преобразуется (неявно) в любой другой тип.

Интересно отметить, что ни один из современных скриптовых языков не содержит перечислений.

Перечисляемые типы улучшают как читабельность, так и надежность программ. Читабельность повышается непосредственно: в отличие от закодированных (числовых) значений, именованные значения легко распознаются и понимаются.

С точки зрения надежности перечисления языки Ada, C#, F#, Java 5.0 и частично C++ предоставляют два преимущества:

- запрещают применение к перечисляемым переменным арифметических операций (например, нельзя складывать дни недели);
- запрещают присвоение значений, выходящих за пределы определенного диапазона.

Этого нельзя сказать, увы, о перечислениях языка C, поскольку в нем элементы перечисления трактуются как целые числа.

В языке C++ ситуация немного лучше [34]. Числовые значения присваиваются перечисляемым переменным типа лишь тогда, когда они явно преобразуются в их тип. В ходе такого преобразования проверяется вхождение числа в диапазон номеров, которые определены для элементов перечисления.

Целые и вещественные типы

Почти в любом языке программирования присутствуют численные типы данных. Наиболее распространенными являются целый и вещественный типы, поскольку они непосредственно поддерживаются аппаратурой компьютера.

Базовый целый тип в языке Pascal обозначается именем `integer`, в языке C — именем `int`.

Множество значений целого типа образует ограниченное упорядоченное подмножество из бесконечного множества математических целых чисел. Максимальное значение иногда задается как именованная константа. Например, в языке Pascal имеется константа `maxint`. В таком случае все допустимые значения целых чисел находятся в диапазоне между `-maxint` и `+maxint`. В языке C существуют четыре разновидности целого типа: `int`, `short`, `long` и `char`, отличающиеся разрядностью чисел и диапазоном значений (например, для типа `int` — разрядность 16 битов [один разряд под знак и 15 разрядов под значение], диапазон значений –32767 ... +32767).

Операции над целочисленными объектами данных обычно разделяются на следующие группы:

- *Арифметические операции*. Бинарные арифметические операции имеют сигнатуру:

  ```
  БинОп : целое × целое → целое
  ```

  Здесь `БинОп` может быть сложением (`+`), вычитанием (–), умножением (*), делением (/ или `div`), а также делением по модулю (`mod`). Сигнатура унарных операций записывается в виде:

  ```
  УнарОп : целое → целое
  ```

где УнарОп может быть, например, операцией смены знака (–). Обычно включаются и другие арифметические операции, часто в виде библиотечных функций (например, операция определения *абсолютного значения*).

- *Операции сравнения*. Сигнатура операции сравнения выглядит следующим образом:

```
СравОп : целое × целое → булево
```

Здесь под СравОп могут подразумеваться следующие операции: равно, не равно, больше чем, меньше чем, больше или равно, меньше или равно. Эти операции сравнивают значения двух операндов и в качестве результата возвращают логическое (булево) значение.

- *Операции присваивания*. Присваивание для целых чисел соответствует одному из двух способов:

```
присваивание (:=) : целое × целое → пустой тип
```

или

```
присваивание (=): целое × целое → целое
```

- *Битовые операции*. В языках с небольшим набором элементарных типов целые числа нагружаются дополнительными «обязанностями». Например, в языке C они используются в качестве булевых величин. Поэтому битовые операции также определены через целые числа:

```
БитОп : целое × целое → целое
```

В C имеются операции для побитовых операций логического умножения (&), логического сложения (|), сдвига битов (<<) и т. д.

- *Логические операции*. В языке C для двух целых чисел, моделирующих значения «ложь» и «истина», заданы две бинарные и одна унарная логические операции:

```
ЛогИ (&&)  : целое × целое → целое
ЛогИли (||) : целое × целое → целое
ЛогНе (!)  : целое → целое
```

Как правило, целочисленный тип данных реализуется при помощи аппаратного представления целых чисел и набора машинных операций, встроенных в процессор.

Базовый вещественный тип в языке Pascal обозначается именем `real`, в языке C — именем `float`. Как и в случае целого типа, допустимые вещественные значения формируют в диапазоне, обусловленном возможностями аппаратуры (от минимального отрицательного значения до максимального положительного). Правда, в отличие от целых чисел, вещественные числа распределены в этом диапазоне неравномерно.

Хранение вещественных чисел с плавающей точкой в памяти компьютера также основано на их аппаратном представлении. Общепринятый формат вещественных чисел с плавающей точкой задает стандарт IEEE 754, этот формат для 32-разрядного числа имеет вид.

Знак (S)	Экспонента (E)	Мантисса (M)
1	8	23

Число представляется с помощью трех полей:

- `S` — однобитовое поле, хранящее знак числа. Значение 0 соответствует положительным числам.
- `E` — экспонента со сдвигом на величину 127. Под это поле отведено 8 битов, таким образом, имеется 256 различных значений от 0 до 255, соответствующих степеням 2 в диапазоне от –127 до 128 (с учетом сдвига).
- `M` — мантисса, под которую отведено 23 бита. Так как в нормализованной форме представления вещественных чисел первый бит в мантиссе всегда равен 1, его можно опускать и вставлять автоматически аппаратными средствами, тем самым увеличивая точность до 24 битов.

Формат обеспечивает диапазон значений чисел от 10^{-38} до 10^{+38}. Для 64-разрядного числа поле под экспоненту расширяется до 11 бит, а поле под мантиссу — до 52 битов, что приводит к представлению чисел в диапазоне от 10^{-308} до 10^{+308}.

Для вещественных чисел обычно предусмотрены те же операции, что и для целых, хотя набор операций сравнения уменьшают. Из-за проблем округления проверку на равенство двух вещественных чисел выполнить не удается, поэтому такая операция обычно запрещается.

Полный набор операций числовых типов для языка Pascal приведен в табл. 9.1, а для C — в табл. 9.2.

Таблица 9.1. Набор операций в языке Pascal (целый, вещественный и логический типы)

Операции	Пояснения
<, <=, =, < >, >=, >, in	Операции отношений и проверки принадлежности
+ , – , or	Аддитивные операции
*, /, div, mod, and	Мультипликативные операции
not	Логическое отрицание

Заметим, что из-за низкого приоритета операций отношения в составных условиях программ на языке Pascal достаточно часто приходится применять скобки. Например, в следующем выражении:

```
(y >= 0) and (x <> M[i])
```

В языке C принята другая схема приоритетов операций. Благодаря этой схеме скобки в составных условиях здесь используются значительно реже. Например, предыдущее условие можно записать без применения скобок группировки:

```
y >= 0 && x != M[i]
```

Таблица 9.2. Набор операций в языке C

Операции	Пояснения
||	Логическое ИЛИ (короткая схема вычисления)
&&	Логическое И (короткая схема вычисления)
= = , !=	Проверка эквивалентности, проверка неэквивалентности
< , <= , > , >=	Операции отношений

Операции	Пояснения
+, –	Аддитивные операции
*, /, %	Мультипликативные операции (операция % вычисляет остаток от целочисленного деления)
!	Логическое отрицание

Короткая схема вычисления, принятая в языке C, означает, что второй операнд обрабатывается только при необходимости:

- Если `E1 = истина`, то `E1 || E2` уже равно `1` (`E2` не вычисляется).
- Если `E1 = ложь`, то `E1 && E2` уже равно `0` (`E2` не вычисляется).

Десятичные числа

Аппаратура большинства компьютеров чаще всего обрабатывает двоичные целые числа и числа с плавающей точкой, однако во многих коммерческих приложениях требуется специальный вид десятичных чисел. Например, объекты данных, представляющие денежные суммы, должны содержать рубли и копейки, то есть выглядеть как десятичные числа с фиксированным количеством десятичных знаков и десятичной точкой. Эта точка располагается в определенном месте и отделяет целую часть числа от дробной. С помощью целых чисел такая запись невозможна, а использование чисел с плавающей точкой может привести к ошибкам округления. Для подобных целей удобны десятичные вещественные числа с фиксированной точкой. Типы данных для них предусмотрены в языках Ada, C# и F#.

Десятичные числа способны представлять точные значения десятичных величин в некотором ограниченном диапазоне. Ограничение диапазона и повышенные затраты на сохранение в памяти считаются их главными недостатками.

В памяти компьютера десятичные числа сохраняются в виде двоичных кодов десятичных цифр. Такие представления получили название двоично-кодированных десятичных чисел (BCD — Binary Coded Decimal). Обычно применяют один из двух вариантов упаковки десятичных цифр:

1) одна цифра на байт памяти;
2) две цифры на байт памяти.

В любом варианте для запоминания числа требуется несколько больше памяти, чем при двоичном представлении. Например, для хранения шести десятичных цифр в BCD-формате (экономичный вариант) необходимы 24 бита памяти, а двоичном формате — 20 битов памяти. Операции над десятичными величинами легко реализуются современными процессорными средствами.

Поддиапазоны

Поддиапазоны представляют собой подтипы некоторых базовых типов, которые имеют счетное количество значений.

Поддиапазоны ограничивают диапазон значений существующего базового типа:

```
0 .. 99
```

Значениями базового типа должны быть целые числа (или другие перечисляемые значения). Поддиапазоны вещественных чисел не допустимы. К поддиапазону применяются те же операции, что и к базовому типу.

Достоинства поддиапазонов:

1. *Уменьшаются затраты памяти.* Поскольку при применении поддиапазона необходимо представлять лишь часть возможных значений базового типа, для хранения элементов этого подтипа требуется меньшая емкость памяти. Например, для представления целых чисел из поддиапазона `1..12` требуется только четыре бита, в то время как для стандартного представления целых чисел может потребоваться 16, 32 или более битов. Тем не менее, так как арифметические операции с укороченными целыми числами могут потребовать программных процедур (то есть будут замедлять вычисления), при реализации таких чисел ориентируются на минимальную разрядность операндов, для которой предусмотрена аппаратная поддержка арифметических операций. Обычно это восемь или шестнадцать разрядов. Например, в C символы хранятся как 8-разрядные целые числа, действия над которыми можно выполнить при помощи машинных команд.
2. *Улучшается контроль переменных.* Если переменная объявлена как экземпляр некоторого поддиапазона, то проверка значений, которые присваиваются этой переменной, становится более точной. Например, если переменная `day_Of_Month` объявлена как `day_Of_Month: 1..31`, то оператор присваивания

   ```
   day_Of_Month := 32;
   ```

   который хотел предложить барон Мюнхгаузен, будет признан ошибочным. Это обнаруживается во время компиляции. Если же переменная `day_Of_Month` объявлена просто как целая, то приведенный оператор присваивания формально оказывается правильным, и программисту придется отыскивать эту ошибку уже на этапе выполнения программы.

Логический тип

В большинстве языков предусмотрен *логический* (булев) тип данных для представления и обработки значений `истина` и `ложь`.

Логический тип данных задает объекты, которые могут принимать два значения: истина (`true`) и ложь (`false`). В языке Pascal логический тип данных рассматривается как встроенное в язык перечисление, а именно:

```
type Boolean = (false, true);
```

В этом объявлении определены константы `false` и `true`, а также соотношение их величин `false < true`.

В обычный набор операций с этим типом данных входят операция присваивания и следующие логические операции:

```
and: Boolean × Boolean → Boolean (И, логическое умножение)
or: Boolean × Boolean → Boolean (включающее Или)
not: Boolean → Boolean (логическое отрицание или булево дополнение)
```

Иногда добавляют и другие логические операции: `эквивалентность`, `исключающее Или`, `импликация`, `not-and`, `not-or`.

Объекты логического типа могут представляться в памяти одним битом (если не учитывать дескриптор, описывающий тип). Поскольку один бит в памяти не имеет отдельного адреса, для представления логического объекта применяют слово или байт (иначе говоря, минимальную адресуемую область памяти). Тогда внутри такой области значения `true` и `false` могут быть обозначены двумя способами:

1) какой-то определенный бит внутри области памяти (например, бит знака числа) используется для представления булева значения (`false = 0, true = 1`), а остальные биты игнорируются;
2) значению `false` соответствуют нули во всех битах области, а любые другие комбинации считают истинным значением.

В языке Java логический тип определен явно, а в языке C он моделируется целыми числами (`истина` соответствует любому ненулевому значению, а `ложь` — нулю).

Если вы работаете с языком C, помните о практическом правиле обеспечения надежности.

Для представления логического значения «истина» следует использовать целочисленную единицу и не стоит пытаться упаковать несколько булевых значений в одно многоразрядное целое число!

Символьные типы

Большая часть данных вводится и выводится из компьютера в символьной форме. Несмотря на то что затем эти данные переводятся в формат других типов, возможность обработки символьных данных в их исходной форме также весьма привлекательна.

Символьный тип данных позволяет создавать объекты данных, значениями которых может быть какой-либо единственный символ. Как уже говорилось, множество возможных значений (символов) обычно является встроенным в язык перечислением. До недавнего времени наиболее распространенной системой кодировки символов считался 8-разрядный код ASCII (American Standard Code for Information Interchange — Американский стандартный код обмена информацией), в которой для 128 различных символов используется диапазон значений 0–127. Другой способ 8-разрядного кодирования задает стандарт ISO 8859-1, обеспечивающий 256 различных символов. Этот стандарт применяется в языке Ada. Впоследствии для поддержки современного информационного пространства был разработан 16-разрядный набор символов по имени Unicode, описанный нами в главе 6. В этом наборе содержатся символы большинства естественных языков мира, а первые 128 символов идентичны символам ASCII. Первым стал использовать набор Unicode язык Java, сейчас он применяется в JavaScript, Python, Perl, C# и F#.

Значения символьного типа упорядочены. Помимо операций отношения и присваивания, в символьных типах иногда применяют операции проверки принадлежности некоторого символа к какой-либо категории символов: буквам, цифрам или специальным символам.

Символьный тип данных обычно прямо поддерживается аппаратурой компьютера, поскольку экземпляры этого типа используются для ввода-вывода. Если же

набор символов языка отличается от набора символов для аппаратуры компьютера, то в языке предусматриваются способы взаимного преобразования.

Стиль программирования в языке C. Преобразование типов

Программа в этом языке оформляется как функция со стандартным именем `main`:

```
#  include  <stdio.h>
int main( )
{
    int  c;
    c = getchar( );
    while  ( c != EOF )
    {
        putchar(c);
        c = getchar( );
    }
}
```

Первая строка — это директива на подключение библиотеки ввода-вывода из заголовочного файла `stdio.h`. Она адресована препроцессору.

Цель программы — читать символы, вводимые с клавиатуры, и выводить их на экран.

Для чтения используют функцию `getchar( )`, для вывода — функцию `putchar( )`. В ходе обработки выполняется *неявное преобразование* символа в целое число:

```
c = getchar( );
```

Необходимость преобразования вызвана тем, что целочисленная константа `EOF` не принадлежит к символьному типу `char`. Обычно функция `getchar( )` возвращает символьное значение, но при достижении маркера конца ввода `EOF` функция считывает целое число.

В языке Pascal преобразование между символами и целыми числами должно задаваться явно:

- `ord(c)` — отображение символа `c` в число *i*.
- `chr(i)` — отображение числа *i* в символ `c`.

Подробное обсуждение преобразований типов приводится в главе 12.

Контрольные вопросы и упражнения

1. Написать четыре различных оператора C, которые прибавляли бы единицу к целой переменной `x`.
2. Написать операторы C, выполняющие следующие действия:
   а) присвоение суммы `x` и `y` переменной `z` и после этого вычисления увеличение значения `x` на единицу;
   б) умножение значения переменной `product` на два с использованием операции `*=`;

в) умножение значения переменной `product` на два с использованием пары операций: `=` и *;
г) сравнение значения переменной `count` с числом 10. Если значение больше, то на экране напечатать сообщение «count больше, чем 10»;
д) уменьшение значения переменной `x` на единицу и затем ее вычитание из переменной `total`;
е) добавление переменной `x` к переменной `total`, а затем уменьшение `x` на единицу;
ж) вычисление остатка от деления `q` на `divisor` и присвоение результата переменной `q`. Записать два разных варианта такого оператора;
з) печать числа 123.4567 с точностью в два знака. Какой ответ будет на экране?

3. Используя язык С, выполнить следующие действия:

а) объявить переменные `sum` и `x` целого типа;
б) инициализировать переменную `x` значением единица;
в) инициализировать переменную `sum` значением ноль;
г) добавить `x` к `sum` и присвоить результат `sum`;
д) увеличить `x` на единицу;
ж) вывести на экран «сумма равна...» и значение самой переменной `sum`.

4. Используя операторы из предыдущего задания, написать программу, которая вычислит сумму целых чисел от одного до десяти. Использовать цикл `while`. Цикл завершается при `x` равном одиннадцати.

5. Определить значение каждой переменной после выполнения вычисления (начальные значения переменных равны пяти).

```
а) product *= x++
б) result = ++x + x
```

Чему равны значения переменных `product` и `x` в первом случае? А во втором случае — чему равны значения `result` и `x` ?

6. Найти и исправить ошибки.

```
а) while (c <= 5) {
     product *= c;
     ++c;
б) while (z >= 0)
     sum += z;
```

Глава 10
Составные типы данных

Десятая глава освещает широкий круг составных типов данных: массивы, строки символов, записи, множества, кортежи и списки. Массивы рассматриваются самым пристальным образом. В частности, обсуждаются следующие вопросы: разновидности массивов (с позиций размещения в памяти и времени жизни), их инициализация, атрибуты и операции простых массивов и их развитых (например, в скриптовых языках) версий. Многомерные массивы подразделяются на прямоугольные и массивы массивов, поясняется специфика доступа к сечениям массивов, а также доступ по содержанию к ассоциативным массивам. Помимо обычных записей описываются объединения и вариантные записи, анализируется их надежность.

Массивы

Массив является наиболее популярной и востребованной структурой данных.

Массив — это структура данных, которая содержит последовательность элементов одинакового типа. Фундаментальное свойство массива — время доступа к любому его элементу A[*i*] не зависит от значения *индекса i*.

Индекс первого элемента называют *нижней границей*, а индекс последнего элемента — *верхней границей* массива.

Тип массива в языке Pascal записывается в виде:

```
type
array [<границы>] of  <тип_элемента>
```

где `[<границы>]` определяют границы массива, а `<тип_элемента>` задает тип элемента массива.

В языке Pascal границы массива задаются перечислением или поддиапазоном:

```
array [1999..2012] of  real -- индексы записаны в виде поддиапазона
array [(пон, втр, срд, чтв, птн)] of  integer -- индексы записаны в виде
                                                 перечисления
array [(char)] of  лексема -- индексы записаны в виде перечисления с именем
```

В большинстве языков программирования границы массива, а значит и индексы, записываются в квадратных скобках. Из современных языков исключением стал лишь язык Ada — в нем используются круглые скобки и объявления типов массива имеют, например, следующий вид:

```
type
array (Day) of Integer; -- индексы записаны в виде перечисления с именем Day
array (Integer range 1999..2012) of  Float; -- индексы записаны в виде диапазона
                                            целых чисел
```

Различают *компоновку* массива (layout) и *размещение* массива в памяти (allocation). При компоновке задаются относительные адреса элементов массива (относительно адреса 1-го элемента), вычисляются формулы для адресации элементов. При размещении определяются действительные машинные адреса элементов и выделяется память.

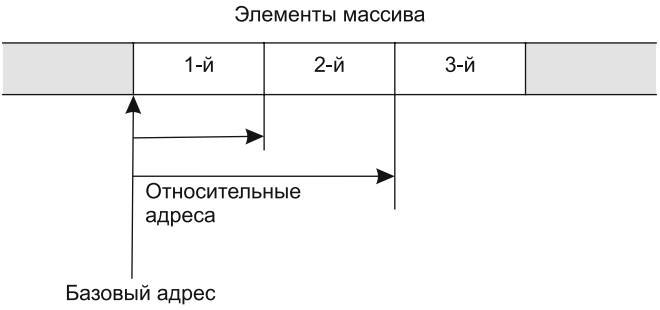


Рис. 10.1. Адресация элементов массива

Как показано на рис. 10.1, возможность использования относительных адресов обусловлена спецификой организации и размещения элементов массива: элементы (ячейки) массива имеют одинаковый размер, а в памяти размещаются плотно, примыкая друг к другу без всяких пропусков.

Объявление массива:

```
var M: array [low .. high] of T;
```

определяет размещение элементов массива в последовательности ячеек памяти. Эти ячейки имеют смежные адреса.

Если `base` — адрес начала массива, `w` — длина элемента массива,
то адрес *i*-го элемента массива равен:

```
A [i] = (base – low × w) + i × w
```

где `(base – low × w)` — вычисляется предварительно, а $(i \times w)$ — вычисляется во время выполнения программы (поскольку *i* изменяется).

В языке C первый элемент массива имеет индекс `0`, поэтому адрес *i*-го элемента массива вычисляется по упрощенной формуле:

```
A [i] = base + i × w
```

Легко заметить, что количество команд машинной программы, необходимых для вычисления адреса, не зависит от значения *i*. Поэтому время доступа к элементу массива — константа.

В конструировании типа «массив» задействуются два вспомогательных типа: один из них определяет тип элемента, а другой — тип индексов. В языках Pascal и Ada для создания индексов применяют многие элементарные типы (целый, логический, символьный и перечисления). Во всех остальных языках разрешены лишь поддиапазоны целых чисел.

При обращении к массиву желательно проверять значение текущего индекса. Значение должно находиться в пределах допустимого диапазона. Такая проверка способствует повышению надежности вычислений. Однако реализована она только в языках Pascal, Ada, Java и C#. В других языках (например, в C и C++) проверка отсутствует в силу специфики организации массивов. Эту специфику мы будем обсуждать в следующей главе.

Несколько необычно организована индексация в языке Perl. Здесь имена всех массивов должны начинаться с символа @, но поскольку элементы массива являются скалярными величинами, их имена предваряются знаком доллара $. Из-за этого в массиве `@list` , например, ко второму элементу обращаются по имени `$list[1]`.

На элемент массива в Perl можно сослаться при помощи отрицательного индекса. Отрицательное значение индекса обозначает номер позиции элемента с конца. Индекс –1 соответствует последнему элементу массива, –2 — предпоследнему и т. д. Для получения индекса, отсчитываемого от начала, нужно отрицательный индекс сложить с размером массива. Например, если массив `@list` состоит из семи элементов с индексами 0..6, то индексированное имя `$list[-2]` адресует предпоследний элемент с индексом `5`. Ссылка на несуществующий элемент массива в Perl возвращает значение `undef`, но сообщение об ошибке не формируется.

Разновидности массивов

Связывание типа индекса массива с переменной массива обычно выполняется статически, но по диапазону индексов связывание происходит иногда динамически.

В некоторых языках нижняя граница диапазона значений индексов задается по умолчанию. Например, в C-подобных языках нижняя граница всех диапазонов индексов равна нулю. В языке Fortran 95 (и выше) она зафиксирована как единица, но может принимать любое целое значение. В большинстве других языков диапазоны индексов полностью определяются программистом.

Классифицируем массивы по трем признакам:

- связывание по диапазонам индексов;
- связывание с памятью;
- категория памяти для размещения.

Всего возможны пять разновидностей массивов. В первых четырех разновидностях связывание по диапазону индексов и размещению в памяти сохраняется в течение всего времени жизни переменной. Имеется в виду, что при фиксации диапазонов индексов массив уже не может изменить размер.

Статические массивы

Статическим называют массив со статическим связыванием по диапазонам индексов и таким размещением в памяти, которое происходит до начала работы программы. Статические массивы достаточно эффективны, поскольку отсутствуют затраты времени как на динамическое размещение в памяти, так и на удаление из нее. Недостатком является то, что память ими занимается на все время выполнения программы.

Явные стековые массивы

Явным стековым называется массив со статическим связыванием по диапазонам индексов и таким размещением в памяти, которое происходит по итогам обработки объявления в ходе выполнения программы (подпрограммы). Массив удаляется из стековой памяти по завершению работы программы (подпрограммы). В сравнении со статическими явные стековые массивы повышают эффективность использования памяти: большой массив одной подпрограммы может занимать ту же область памяти, что и большой массив другой подпрограммы, причем так может продолжаться до тех пор, пока обе подпрограммы не окажутся активными одновременно. К недостатку следует отнести дополнительные затраты времени на размещение и удаление массива из памяти.

Стековые массивы

Стековым называется массив с динамическим связыванием по диапазонам индексов и динамическим размещением в памяти, которое происходит в ходе обработки объявления. Связанность диапазонов индексов и размещение массива в памяти сохраняются в течение всей жизни переменной. Главным преимуществом этой разновидности массивов по сравнению с двумя предыдущими считается гибкость — не нужно заранее знать (до момента использования) размер массива.

Явные динамические массивы

Явный динамический массив подобен явному стековому массиву в том, что связывание и по диапазонам индексов, и по памяти происходит после размещения в памяти. Связывание по индексам и памяти производится по запросу программы и в течение ее выполнения, но память выделяется в куче, а не в стеке. Преимуществом этой разновидности массивов является гибкость: изменение размера массива всегда остается проблемой. Недостаток — на размещение в куче тратится времени больше, чем на размещение в стеке.

Динамические массивы

Динамическим называется массив с таким динамическим связыванием по диапазонам индексов и размещению в памяти, которое повторяется многократно в течение всего жизненного цикла массива. Преимущество: максимальная гибкость. Массив по мере необходимости может расти или сжиматься прямо в ходе выполнения программы. Недостаток — на многократное размещение (удаление) в куче тратится много времени (и все оно приходится на период вычислений).

В языках C и C++ статическими являются массивы, которые объявляются в функциях со спецификатором `static`.

Массивы, объявленные в функциях C и C ++ без спецификатора `static`, считаются явными стековыми массивами.

Как показано ниже, массивы языка Ada могут быть стековыми:

```
get(Size);
declare
    Vector : array  (1 .. Size) of  Integer;
begin
…
end;
```

В этом фрагменте с помощью процедуры `get` вводится размер `Size` для массива `Vector`, который затем динамически размещается в памяти при переходе к блоку `declare`. Когда выполнение доходит до закрывающей скобки блока, массив `Vector` из памяти удаляется.

Языки C и C++ также обеспечивают явные динамические массивы. В этом случае используются стандартные библиотечные функции `malloc` и `free`, которые являются операциями для размещения в куче и удаления из нее соответственно. В языке C++ для управления кучей применяют операторы `new` и `delete`. Имя массива здесь рассматривается как указатель на набор ячеек памяти; указатель может индексироваться.

В языке Java все не родовые массивы считаются явными динамическими. После создания эти массивы сохраняют связывание по диапазонам индексов и памяти. Язык C# также обеспечивает данную разновидность массивов.

Кроме того, язык C# обеспечивает родовые динамические массивы, которые являются объектами класса `List`. Первоначально эти объекты-массивы считаются пустыми и создаются оператором:

```
List<String> stringList = new List<String>();
```

Элементы к объекту добавляются методом `Add`:

```
stringList.Add("Liza");
```

Доступ к элементам этих массивов организуется через индексирование.

Язык Java содержит родовой класс `ArrayList`, подобный классу `List` из языка C#. Разница лишь в том, что здесь индексирование не применяется — для доступа к элементам должны использоваться методы `get` и `set`.

В языке Perl массив может быть увеличен с использованием `push` (добавить один и более элементов в конец массива) и `unshift` (добавить один и более элементов в начало массива) или указанием элемента массива с использованием индекса, значение которого больше последнего индекса массива. Массив может быть сокращен до пустого за счет присвоения ему пустого списка, задаваемого символами `( )`. Длина массива вычисляется сложением значения последнего индекса с единицей.

Подобно Perl, язык JavaScript обеспечивает рост массивов с помощью методов `push` и `unshift`, а также сокращение (установкой массива в состояние пустого списка). Отрицательные индексы здесь не поддерживаются.

Массивы в JavaScript могут быть *разреженными*, в которых значения индексов не образуют непрерывную последовательность. Рассмотрим массив по имени `list` из 12 элементов и с диапазоном индексов `0..11`. Выполним следующий оператор присваивания:

```
list[70] = 47;
```

Теперь массив `list` имеет 13 элементов и длину `71`. Элементы с индексами `12..69` не определены, для их хранения память не нужна. Ссылка на несуществующий элемент в массивах JavaScript считается неопределенной.

Массивы в языках Python, Ruby и Lua увеличивают или с помощью методов для добавления элементов, или путем соединения с другими массивами. Языки Ruby и Lua поддерживают отрицательные индексы, а Python — нет. В языках Python, Ruby, и Lua элемент или сечение массива можно удалить. Ссылка на несуществующий

элемент в языке Python приводит к ошибке периода выполнения, а в языках Ruby и Lua считается неопределенной (донесение об ошибке не формируется).

В функциональном языке ML массивы не определены, но широко используются в такой реализации языка, как SML/NJ. Массивы функционального языка F# подобны массивам в C#. Для обработки массивов в язык F# включен оператор `foreach`.

Инициализация массива

Некоторые языки предусматривают средства для инициализации массивов во время их размещения в памяти.

В языке Fortran 95 (и выше) массив можно инициализировать, определив в его объявлении агрегат массива. Для одномерного массива агрегат является списком литералов, ограниченных круглыми скобками и слешами (прямыми наклонными чертами). Например, можно написать

```
Integer, Dimension (3) :: List = (/0, 4, 7/)
```

Языки C, C++, Java и C# также обеспечивают инициализацию массивов, но с одной новой особенностью: в языке C объявление

```
int list [] = {3, 4, 8, 79};
```

заставляет компилятор самостоятельно установить длину массива. Это удобно, но вносит ограничения: такая инициализация лишает систему возможности обнаруживать ошибки выхода индексов за пределы корректного диапазона. Ведь диапазон индексов явно не объявляется!

Символьные строки в языках C и C++ реализованы как массивы из символов типа `char`. Эти массивы могут быть инициализированы строковыми литералами:

```
char surname [] = "orlov";
```

Массив `surname` будет содержать шесть элементов, так как все строки завершаются символом `'\0'`, неявно добавляемым системой к строковым константам.

Массивы строк в языках C и C++ также могут инициализироваться наборами строковых литералов. В этом случае речь идет о массиве указателей на символы. Например:

```
char *towns [] = {"Riga", "Petersburg", "Kiev"};
```

Этот пример высвечивает суть символьных литералов в языках C и C++. Для инициализации символьного массива `surname` мы применили строковый литерал, являющийся массивом типа `char`. Однако в массиве `towns` литералы представляются указателями на их символы, поэтому и сам массив является массивом указателей на символы. Например, `towns[0]` — это указатель на букву `'R'` в массиве литеральных символов, содержащем символы `'R'`, `'i'`, `'g'`, `'a'` и символ нуля `'\0'`.

В языке Pascal инициализация массивов (в разделе объявлений программы) не предусмотрена.

Язык Ada предлагает две разновидности агрегатов для инициализации массивов в операторе объявления:

- *позиционный агрегат*, в котором значения элементов перечисляются в порядке их появления в массиве;

- *именной агрегат*, где с помощью операции-стрелки `=>` показываются непосредственные присваивания значений соответствующим индексным позициям массива.

Приведем примеры:

```
Buffer: array (1..5) of Integer := (1, 5, 6, 8, 9);
Container : array (1..5) of Integer := (1 => 21, 4 => 55, others => 0);
```

В первом объявлении все элементы массива `Buffer` инициализируются значениями, перечисленными в порядке следования элементов в массиве. Во втором объявлении первый и четвертый элементы массива `Container` инициализируются с помощью прямых присваиваний, а имя `others` обозначает все остальные элементы, которые инициализируются нулевыми значениями.

Атрибуты и операции простого массива

Простыми будем называть одномерные массивы фиксированного размера, создаваемые с помощью императивных языков (C, Pascal) или императивных средств таких языков, как C++ и Ada.

Такие массивы однородны, то есть размер и структура каждого их элемента одинаковы. Фиксированная размерность массива означает неизменность количества элементов и их местоположения в течение всего периода его жизни. В объект данных типа массив включается *дескриптор*, описывающий некоторые или все атрибуты массива. Верхняя и нижняя границы диапазона индексов (которые не требуются для доступа к элементам) сохраняются для того, чтобы обеспечить проверку соответствия индекса объявленному диапазону. Другие атрибуты обычно в дескрипторе не хранятся (во время вычислений); они нужны только во время компиляции для контроля типов и определения способа хранения массива в памяти.

В состав атрибутов для объекта данных типа «простой массив» входят:

- *Количество элементов* — указывается косвенно, путем задания диапазона изменения индексов.
- *Тип данных для каждого элемента* — в данном случае он одинаков для всех элементов.
- *Список значений индексов*, применяемых для выбора элементов, — задается в виде набора целых чисел, первое из которых соответствует первому элементу, второе — второму элементу и т. д. Он может представляться в виде диапазона значений, например `[-7..25]`, или определяться только верхней границей диапазона, если нижняя задается по умолчанию, например `[100]`.

Базовой считают операцию выбора элемента массива. Она называется *индексацией* и обозначается в виде имени массива с присоединенным индексом искомого элемента: `B[2]` или `ExamsMark[student]`. В общем случае индекс может записываться как вычисляемое выражение: `ExamsMark[`*i* `+ 2]`. В левой части оператора присваивания операция индексации возвращает *l*-значение искомого элемента (его адрес), а в правой части — *r*-значение искомого элемента (его величину).

Индексация позволяет применять к элементам массивов все операции, которые разрешены для типов этих элементов.

В языках Pascal и Ada возможны присваивания массивов целиком. Дополнительно в языке Ada разрешены операции сравнения и конкатенации простых массивов, а также логические операции над простыми булевыми массивами. При конкатенации (составлении) длина результирующего массива равна сумме длин массивов-операндов.

В языке Fortran 95 (и выше) предусмотрены разнообразные операции над массивами, предназначенные для *поэлементной* обработки пар массивов. К ним относятся присваивания, операции отношений, арифметические и логические операции. При выполнении, например, сложения двух массивов одинаковой размерности попарно складываются значения их элементов.

В C-подобных языках отсутствуют операции, при выполнении которых массив считается единым целым. Исключением являются объектно-ориентированные методы языков Java, C++ и C#.

Операции над массивами в скриптовых языках

В языке Python массивы называются списками, хотя они имеют все характеристики динамических массивов. Поскольку объекты могут иметь любые типы, массивы здесь неоднородны. Python обеспечивает присваивание массива, хотя оно заключается лишь в изменении ссылок. Python также имеет операции для конкатенации массивов (`+`) и определения вхождения (членства) элемента (`in`). Он содержит две различные операции сравнения: одна определяет, ссылаются ли две переменные на один и тот же объект (`is`); другая проверяет равенство всех объектов, входящих в указанные объекты, независимо от глубины вложенности (`==`).

Подобно языку Python, элементы в массивах Ruby являются ссылками на объекты. И здесь операция `==` над двумя массивами возвращает `true`, если массивы имеют одинаковую длину, а соответствующие элементы эквивалентны. Массивы в Ruby могут подвергаться конкатенации с помощью метода класса `Array`.

Язык Perl обеспечивает присваивания массивов, но не поддерживает сравнения.

Прямоугольные массивы и массивы массивов

Прямоугольный массив — это многомерный массив, в котором все строки и столбцы имеют одинаковое количество элементов. Прямоугольные массивы точно моделируют прямоугольные таблицы.

Объявление двухмерного прямоугольного массива

```
var M: array [1 .. 3, 1 .. 2] of integer;
```

можно рассматривать как три двухэлементных подмассива `M[1]`, `M[2]`, `M[3]`, которые являются строками в следующей матрице элементов:

$$\begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \\ M_{31} & M_{32} \end{bmatrix}.$$

Возможны два варианта размещения двухмерного массива в памяти: по строкам и по столбцам.

Размещение массива *по строкам* проиллюстрировано на рис. 10.2.

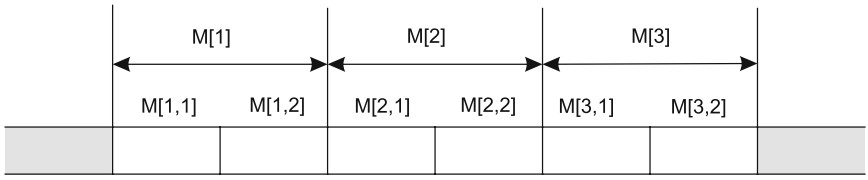


Рис. 10.2. Размещение двухмерного массива по строкам

В этом случае быстрее всего изменяется последний индекс *j* каждого элемента `M[`i,j`]`.

Адрес элемента можно определить по формуле:

$$A[i_1][i_2] = (base - low_1 \times w_1 - low_2 \times w_2) + i_1 \times w_1 + i_2 \times w_2$$

где w_1 — длина строки $M[i_1]$,
w_2 — длина элемента $M[i_1][i_2]$,
low_1 — нижняя граница строки,
low_2 — нижняя граница элемента строки.

Очевидно, что

$$w_1 = n_2 \times w_2$$

где n_2 — количество элементов в строке,

$$n_2 = high_2 - low_2 + 1$$

При размещении массива *по столбцам* быстрее всего изменяется первый индекс *i* каждого элемента `M[`i,j`]`:

```
M[1,1],   M[2,1],   M[3,1],   M[1,2],   M[2,2],   M[3,2]
```

Прямоугольные массивы можно создавать в языках Fortran, Pascal, Ada, F# и C#.

В этих случаях все индексные выражения в ссылках на элемент помещаются в единую пару скобок (круглых — в языках Fortran и Ada, и квадратных — во всех остальных языках). Например, на языке C# можно записать:

```
myArray[3, 7]
```

Массив массивов — это многомерный массив, в котором не требуется, чтобы длины у строк были одинаковыми. Массивы массивов иногда называют *не выровненными* массивами. Например, не выровненная матрица может состоять из трех строк, одна из которых содержит пять элементов, другая — семь элементов, а третья — двенадцать элементов. Все эти соображения применимы и к массивам более высоких размерностей. Так, для трехмерного массива в третьем измерении (измерении уровней) каждый уровень может иметь различное количество элементов. Не выровненные массивы становятся возможны, когда многомерные массивы, по сути, являются массивами, состоящими из массивов. Например, матрица строится как массив из одномерных массивов.

Рассмотрим пример построения не выровненного массива средствами языка C#, в котором элементы подобного массива имеют ссылочный тип и инициализируются значением `null`.

Объявим одномерный массив из трех элементов, каждый из которых является одномерным массивом целых чисел:

```
int[][] jaggedArray = new int[3][];
```

Перед использованием `jaggedArray` его элементы нужно инициализировать. Сделать это можно следующим образом:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Каждый элемент представляет собой одномерный массив целых чисел. Первый элемент массива состоит из пяти целых чисел, второй — из четырех и третий — из двух.

Для заполнения элементов массива значениями можно выполнить инициализацию, при этом размер массива знать не требуется:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Также массив можно инициализировать в объявлениях:

```
    int[][] jaggedArray2 = new int[][]
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

В языке C# можно использовать и сокращенную форму. Следует помнить, что при инициализации элементов оператор `new` опускать нельзя, так как инициализация по умолчанию не предусмотрена:

```
int[][] jaggedArray3 =
{
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

Напомним, что не выровненный массив является массивом массивов и поэтому его элементы имеют ссылочные типы и инициализируются значением `null`.

Доступ к отдельным элементам массива организуется следующим образом:

```
// Присвоить 77 второму элементу ([1]) первого массива ([0]):
jaggedArray3[0][1] = 77;
// Присвоить 88 второму элементу ([1]) третьего массива([2]):
jaggedArray3[2][1] = 88;
```

Массивы массивов можно смешивать с прямоугольными массивами. Покажем объявление и инициализацию одномерного массива массивов, состоящего из трех двухмерных элементов различного размера:

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

Проиллюстрируем возможность доступа к отдельным элементам массива:

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

Здесь выводится на экран значение элемента `[1,0]` первого массива (значение `5`).

Количество массивов, содержащихся в массиве массивов, можно определить с помощью метода `Length`:

```
System.Console.WriteLine(jaggedArray4.Length);
```

В данном случае возвращается значение `3`.

Языки C, C++ и Java поддерживают массивы массивов, но не предусматривают прямоугольные массивы. В этих языках при указании элемента многомерного массива записывается отдельная пара квадратных скобок по каждому измерению:

```
myArray[3][7]
```

Например, в языке C++ массив из чисел с плавающей точкой можно объявить двумя размерами, которые заключены в квадратные скобки:

```
float sales [ districts ][ months ];
```

В этом объявлении выражения `districts` и `months` определяют размеры массива по каждому измерению. Конечно, массивы могут иметь бóльшую размерность. Трехмерный массив — это массив массивов, которые состоят из массивов. Доступ к элементам трехмерного массива осуществляется с использованием трех индексов:

```
elem = dimen3 [ x ][ y ][ z ];
```

Элементы же двухмерного массива требуют двух индексов:

```
sales [ d ][ m ]
```

Еще раз заметим, что каждый индекс заключается в отдельные квадратные скобки. Запятые не используются. Нельзя писать `sales [ d, m ]`; это работает в других языках, но не в C++.

Сечения массивов

Сечение — это подструктура массива, которая сама является массивом. Важно понимать, что сечение не является новым типом данных. Это скорее способ обращения к части массива как к единому целому. Если с массивами языка нельзя обращаться как с единым целым, то в таком языке нет и сечений.

На рис. 10.3 показаны примеры сечений:

- на рис. 10.3, *а* приведено сечение в виде второго столбца матрицы, состоящей из трех столбцов;
- на рис. 10.3, *б* сечение соответствует третьей строке матрицы, состоящей из четырех строк;
- на рис. 10.3, *в* сечением считается третья плоскость трехмерного массива.

Если массив объявлен как `S[4, 3]`, то сечение, изображенное на рис. 10.3, *а*, обозначается через `S1[*, 2]`, где символ * означает, что первый индекс (строка) меняется от 1 до 4. Два других сечения описываются как `S2[3, *]` и `S3[3, *, *]`.

На языке Fortran 90 данные сечения описываются так:

- `S1(1 : 4, 2)`;
- `S2(3, 1 : 3)`;
- `S3(3, 1 : 3, 1 : 4)`.

Сечения могут быть аргументами в операторах вызова подпрограмм.

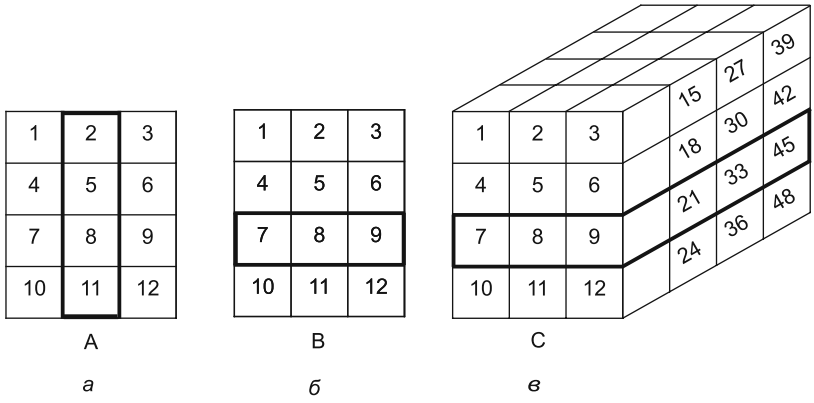


Рис. 10.3. Сечения массивов: *а*) одномерное сечение — столбец; *б*) одномерное сечение — строка; *в*) многомерное сечение — плоскость

Использование дескрипторов позволяет эффективно реализовать сечение. Например, дескриптор матрицы `S` размером 3 на 4 представлен в табл. 10.1.

Таблица 10.1. Дескриптор матрицы S размером 3 на 4

Имя атрибута	Значение
(base – w_1 – w_2)	base – 4
low_1	1
$high_1$	4
w_1	3
low_2	1
$high_2$	3
w_2	1

В этом случае формула вычисления адреса элемента `S[`i`,` j`]`, приведенная в предыдущем разделе, упростится до следующего вида:

$A(S[i, j]) = (base - w_1 - w_2) + i \times 3 + j \times 1$

Заметим, длина элемента массива w_2 определяет расстояние между соседними элементами массива. В данном случае элементы массива следуют непрерывно один за другим. Однако это лишь частный случай; для сечения характерно, что все его элементы одинаково удалены друг от друга, но при размещении массива в памяти они не обязательно «прижаты» друг к другу. Следовательно, дескриптор сечения `S1[*, 2]` (рис. 10.3, *а*) должен иметь описание, представленное в табл. 10.2.

Таблица 10.2. Дескриптор сечения S1[*, 2]

Имя атрибута	Значение
(base – w_1)	base – 3
low_1	1
$high_1$	4
w_1	3

Этот дескриптор описывает одномерный массив, состоящий из четырех элементов, начальная позиция которого смещена на единицу относительно начальной позиции массива `S` и элементы которого разнесены с промежутком, равным трем позициям памяти. В качестве упражнения составьте дескрипторы для сечений `S2[3, *]` и `S3[3, *, *]`.

В языке Ada разрешены лишь сечения, включающие последовательные элементы одномерного массива.

Рассмотрим следующие объявления на языке Python:

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
matrix = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

Вспомним, что начальным индексом в массиве языка Python является 0. Синтаксисом языка Python сечение указывается парой числовых выражений, разделенных двоеточием. Первое выражение задает первый индекс сечения, а второе — первый индекс после последнего индекса в сечении. Поэтому `vector[3:6]` является трехэлементным массивом, содержащим 4-й, 5-й и 6-й элементы массива (их индексами являются 3, 4 и 5). Строка матрицы определяется одним индексом. Например, `matrix[1]` указывает на вторую строку матрицы `matrix`; часть строки можно описать как часть одномерного массива. Например, запись `matrix[0][0:2]` отсылает к первому и второму элементу первой строки матрицы, то есть к `[1, 2]`.

Python также обеспечивает более сложные формы сечения массивов. Например, в записи `vector[0:7:2]` цифра после второго двоеточия определяет, с каким *шагом* следует извлекать символы, поэтому сечение примет вид: `[2, 6, 10, 14]`.

Язык Perl поддерживает две формы сечений: в виде списка определенных индексов или в виде диапазона индексов. Например,

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

Заметим, что ссылки на сечения используют имена массивов, а не скалярные имена, поскольку сечения являются массивами, а не скалярами.

Язык Ruby поддерживает сечения с помощью метода `slice` из его объекта `Array`, который может получать три формы параметров. Параметр «единичное целое выражение» интерпретируется как индекс, при этом метод вернет элемент с данным индексом. Если метод `slice` получает в виде параметров два целых выражения, то первое интерпретируется как начальный индекс, а второе — как количество элементов в сечении. Определим следующий массив `list`:

```
list = [2, 4, 6, 8, 10]
```

Тогда вызов `list.slice(2, 2)` вернет `[6, 8]`. Третьей параметрической формой для метода `slice` является диапазон, который задается как целое выражение, двоеточие и второе целое выражение. При наличии параметра-диапазона метод `slice` вернет массив элементов с данным диапазоном индексов. Например, вызов `list.slice(1..3)` вернет сечение `[4, 6, 8]`.

Статические массивы языка C

В языке C массивы компонуются статически (во время компиляции), а размещаются в памяти во время каждого вызова функций, то есть считаются явными стековыми массивами. Исключением являются статические массивы. В них размещение выполняется статически, до начала работы программы. Элементы статических массивов сохраняют свои значения между вызовами функций. Рассмотрим следующую функцию:

```
int calc ( ) {
    static char buffer [128];
    int moneta [] = {1, 5, 10, 20, 50, 100};
        …
}
```

В ней объявлены два массива. Массив `moneta []` объявлен обычно, его размер и тип элементов система определяет самостоятельно, исходя из анализа количества элементов и их значений, инициализация элементов задана в строке объявления.

Массив `buffer` состоит из 128 элементов символьного типа. Он объявлен как статический массив.

Память для `buffer [128]` выделяется до начала выполнения программы, его элементы сохраняют свои значения между вызовами функции `calc ( )`. Элементы массива `moneta[]` между вызовами функции теряют свои значения и могут менять местоположение.

Ассоциативные массивы

Во всех рассмотренных нами массивах доступ к отдельным элементам выполнялся по индексу. Значение индекса определяло место размещения элемента в массиве.

Иногда на предварительное упорядочение элементов данных в массиве просто нет времени, или же накладные расходы на поддержание порядка очень велики. В таких ситуациях желателен альтернативный подход к хранению и извлечению данных из массива.

Такой подход реализуют ассоциативные массивы.

Ассоциативный массив — это *неупорядоченное* множество элементов данных, индексированных таким же количеством величин, которые называются *ключами*. В обычных массивах индексы никогда не надо запоминать (поскольку они идут по порядку). В ассоциативном же массиве ключи определяются конкретным пользователем и должны содержаться в самой структуре массива. Каждый элемент ассоциативного массива несет в себе ключ и значение. Ключ является единственным средством доступа к значению.

Ассоциативные массивы широко используются в таких языках, как Perl, Python, Ruby и Lua. Кроме того, они обеспечиваются стандартными библиотеками классов в языках Java, C++, C# и F#.

В языке Perl ассоциативные массивы часто называют хешами, так как их элементы записываются в память и извлекаются из нее с помощью функций хеширования. Пространство имен хешей в языке Perl четко обозначено — имя каждой

хешированной переменной должно начинаться со знака процента (`%`). Любой хешированный элемент состоит из двух частей: ключа, представляемого строкой, и значения, являющегося скалярной величиной (числом, строкой или ссылкой). Хешам могут присваиваться литеральные величины, например:

```
%age = ( "Liza" => 27, "Alex" => 18,
    "Nataly" => 30, "John" => 41);
```

Форма обращения к отдельным элементам хеша достаточно своеобразна: вначале записывается модифицированное имя массива (модификация состоит в превращении имени массива в имя скалярной переменной и сводится к замене начального символа `%` на знак доллара `$`), а затем в фигурных скобках указывается ключ. Напомним, что имя скалярной переменной всегда должно начинаться со знака доллара (в языке Perl, разумеется). Приведем пример:

```
$age{"Nataly"} = 33;
```

Новый элемент добавляется к хешу с помощью той же формы оператора. Удаляется элемент из хеша по оператору `delete`:

```
delete $age{"John"};
```

Обнуление хеша выполняется путем присвоения ему пустого литерала:

```
@age = ();
```

Размер хеша в языке Perl изменяется динамически. Хеш увеличивается при добавлении нового элемента и уменьшается при удалении элемента (или при присвоении ему пустого литерала).

Операция `exists` позволяет проверить наличие элемента в массиве. Она возвращает значение `true`, если ее операнд (ключ) присутствует в массиве. В противном случае формируется значение `false`. Например:

```
if (exists $age { "Liza"}) ...
```

Другими операциями над хешами являются:

- операция `keys` — возвращает массив ключей хеша;
- операция `values` — возвращает массив значений хеша;
- операция `each` — возвращает полную информацию по всем элементам (ключ плюс значение).

Ассоциативные массивы в Python, называемые *словарями*, подобны хешам Perl, но все их элементы являются ссылками на объекты. Ассоциативные массивы в Ruby похожи на их аналоги в Python, но ключи в них могут быть любыми объектами, а не только строками. Таким образом, отчетливо прослеживается линия развития: от хешей Perl, где ключи должны быть строками, через массивы PHP, где ключи могут быть целыми или строками, к хешам в Ruby, в котором ключом может быть объект любого типа.

В PHP применяются как обычные, так и ассоциативные массивы. Они могут трактоваться двояко. Язык обеспечивает функции как индексированного, так и хешированного доступа к элементам. Массивы могут иметь элементы, которые создаются с помощью простых числовых индексов, а также элементы, создаваемые посредством строковых ключей хеширования.

В языке Lua единственной структурой данных является таблица. Таблица Lua — это ассоциативный массив, в котором и ключи, и значения могут быть любого типа.

Таблица может использоваться как обычный массив, как ассоциативный массив или как запись (структура). В «режиме» обычного или ассоциативного массива ключи помещаются в квадратные скобки. Если таблицу применяют как запись, ключи считаются именами полей и в ссылках на поля можно использовать нотацию «точка»: `имя_записи.имя_поля`.

Языки C# и F# поддерживают ассоциативные массивы посредством класса .NET.

Для поиска элемента ассоциативный массив значительно лучше обычного массива, поскольку неявные операции хеширования, применяемые для доступа к элементу хеша, очень эффективны. Более того, ассоциативные массивы идеально подходят для случая, когда надо сохранять парные данные, например имя и зарплату служащего. С другой стороны, для обработки каждого элемента списка лучше использовать обычные массивы.

Строки символов

Строка — это просто последовательность символов. Для человека строка является наиболее естественной формой представления информации, в силу чего она играет существенную роль во всех языках программирования.

В языках программирования строки представляются или в виде отдельного элементарного типа, или в виде массивов из элементов символьного типа.

Различают три разновидности строк:

1) строки фиксированной длины (строки со статической длиной);
2) строки переменной длины, не превосходящей заданного максимума (строки с ограниченной динамической длиной);
3) неограниченные по длине строки (строки с неограниченной динамической длиной).

Обычно на аппаратном уровне поддерживаются только строки фиксированной длины, а для реализации других разновидностей требуется программное моделирование, которое чаще всего обеспечивает компилятор, реже — программист.

Объявление строковой переменной фиксированной длины вводит следующие правила:

- значениями могут быть строки символов только этой длины;
- присваивание строки символов с длиной, отличной от заданной, приводит к ее усечению (если длина больше заданной) или к добавлению пробелов в ее конец (если длина меньше заданной).

К данной разновидности относятся строки в языке Python, неизменяемые объекты класса `String` в Java, а также строковые классы из стандартной библиотеки классов C++, встроенный класс `String` в Ruby, средства из библиотеки классов .NET, доступные в языках C# и F#.

Правила работы со строками с ограниченной динамической длиной имеют вид:

1) в программе указывается максимальная длина строки;
2) фактические строки могут иметь меньшее количество символов;
3) текущая длина строки может меняться, но при превышении максимума лишние символы отсекаются.

Подобные строки приняты в языках C и C++ (строки в стиле C). Правда, вместо объявления длины строки здесь используется ограничивающий нуль-символ `\0`, который помещается за последним символом строки.

При обработке строк с неограниченной динамической длиной обеспечивается максимальная гибкость, но существенно возрастают затраты на их динамическое размещение в памяти, а также на удаление из памяти. Такие строки применяют в JavaScript, Perl и стандартной библиотеке C++.

Язык Ada поддерживает все три разновидности строк.

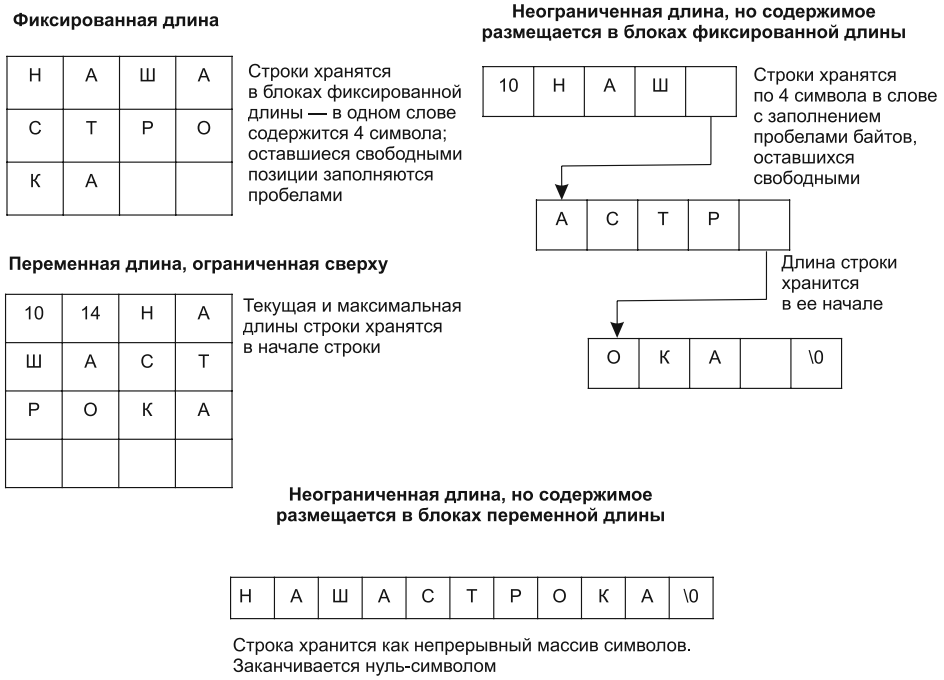


Рис. 10.4. Размещение строк в памяти

Каждая из трех разновидностей строк по-своему размещается в памяти компьютера (рис. 10.4). Для строк с ограниченной динамической длиной дескриптор описывает максимальную и фактическую длину строки, находящейся в объекте данных. Строки с неограниченной динамической длиной содержатся или в связанной цепочке объектов фиксированной длины, или в непрерывном массиве символов. Непрерывные массивы часто применяют в языке C++.

Первые две разновидности строк позволяют отвести под каждый объект этого типа фиксированную область памяти еще при компиляции. Если же длина строк не ограничена, необходимо динамически распределять память для таких объектов уже в процессе выполнения программы. Разные способы представления строк приводят к различным наборам операций над строками. Охарактеризуем наиболее важные операции.

Конкатенация (*объединение*). Конкатенацией называют операцию, объединяющую две короткие строки в одну более длинную. Например, если символ `&` обо-

значает конкатенацию, то выполнение операции `"Язык " & "программирования"` приведет к строке `"Язык программирования"`.

Операции отношений над строками. К строкам можно применять обычные операции отношения: `равно`, `больше-чем`, `меньше-чем`. Применительно к строкам, алгоритмы выполнения этих операций основываются на правиле лексикографического (алфавитного) упорядочения символов: строка *X* меньше строки *У*, если первый символ в *X* меньше первого символа в *У*; если эти символы совпадают, то второй символ *X* должен быть меньше второго символа У. Этот процесс распространяется на все символы строк. Если же одна из сравниваемых строк длиннее, то вторую строку удлиняют за счет пробелов так, чтобы их длины стали одинаковыми.

Выделение подстроки при помощи индексов. В этом случае предполагается, что строка представляется как символьный массив. Результатом операции становится набор символов, позиции которых указаны индексами, являющимися аргументами операции: первый индекс задает первую позицию, а второй — последнюю позицию подстроки. Например, в языке Ada имеется встроенный тип `String`, который определяет строки как одномерные массивы элементов, принадлежащих к типу `Character`. Объявление этого типа имеет вид:

```
type String is array (Positive range <>) of Character;
```

где указано, что индексами могут быть положительные целые числа (типа `Positive`), а диапазон индексов не ограничен (об этом говорит фраза `range <>`).

Для типа `String` предусмотрены: операция выделения подстроки, конкатенация, операции отношений и присваивание.

Объявим две строки, состоящие из 5-ти и 9-ти символов (для 9-символьной строки определим начальное значение):

```
Alias : String ( 1 .. 5 );
Name : String ( 1 .. 9 ) := "Aleksandr";
```

Теперь мы можем выделить пять символов из строки `Name` и присвоить их строке `Alias`, в результате чего она получит значение `"Aleks"`:

```
Alias := Name ( 1 .. 5 );
```

Выделение подстроки на основе сопоставления с образцом. Иногда точная позиция требуемой подстроки неизвестна, но известны какие-то характеристики подстроки. Например, она располагается после символа `'a'`, или после десятичной точки, или после фразы `"Пароль"`. Одним из аргументов операции сопоставления с образцом является *образец* — специальная структура данных, которая определяет вид искомой подстроки (например, некоторые символы, цифры, специальные знаки). Второй аргумент операции сопоставления с образцом указывает исходную строку, в которой будет выполняться поиск подстроки, соответствующей заданному образцу. Очень часто образец записывается в виде регулярного выражения.

Правила создания регулярного выражения:

1. Регулярными выражениями являются отдельные терминальные символы.
2. Если `a` и `b` — регулярные выражения, то `a∨b`, `ab`, `(a)` и `a*` также являются регулярными выражениями.
3. Заключение регулярного выражения в круглые скобки приводит к созданию группы, к которой можно применить другие операции, перечисленные в пункте 2.

Здесь обозначено:

- `ab` — результат конкатенации, или объединения, в последовательность регулярных выражений `a` и `b`;
- `a∨b` — выбор одного из выражений `a` или `b`;
- `a*` — замыкание Клини регулярного выражения `a` (символ «*» обозначает операцию итерации), которым является ноль или более повторений регулярного выражения `a`. Примеры: пустая строка, `a`, `aa`, `aaa`,... и т. д.

ПРИМЕЧАНИЕ

На практике вводят дополнительные соглашения по записи регулярных выражений. Например, используют запись p^+ для сокращенного обозначения выражения pp*. Операцию «$^+$» называют позитивным замыканием или усеченной итерацией.

Например, образец c^+d^+, записанный в виде регулярного выражения, определяет цепочку символов, начинающуюся с одного или более символов `c`, за которыми обязательно следует один или более символов `d`.

Кроме описанных операций над строками, как правило, определяется большое количество операций форматирования. Они обеспечивают форматирование строк при выполнении ввода-вывода.

В языках C и C++ для хранения строк символов используются массивы с элементами типа `char`, а набор операций со строками предусмотрен в стандартной библиотеке языка C. Библиотечные функции, создающие строки, сами вносят в них ограничивающий нуль-символ `\0`. Рассмотрим следующее объявление:

```
char area[] = "world";
```

Здесь создан массив `area` из шести символьных элементов, причем последним элементом является нуль-символ `\0`. Кроме воспроизводства C-строк, язык C++ дополнительно поддерживает строки с помощью своей стандартной библиотеки классов. Обычно программисты на C++ предпочитают работать со строками с помощью библиотечного класса `string`, не прибегая к символьным массивам и библиотеке языка C.

В языке Java строки поддерживаются классом `String`, обеспечивающим константные строки, и классом `StringBuffer`, ориентированным на изменяющиеся строки, более похожие на массивы отдельных символов. Такие строки формируются методами класса `StringBuffer`. Языки C# и Ruby содержат строковые классы, подобные классам Java.

В языке Python все строки принадлежат к элементарному типу с операциями выделения подстрок, конкатенации, индексированного доступа к отдельным символам, а также методами для поиска и замещения. Имеется также операция проверки принадлежности символа к некоторой строке. Таким образом, несмотря на то что строки в Python принадлежат к элементарному типу, в отношении поиска и выделения подстрок они очень похожи на массивы символов. Тем не менее строки в языке Python считаются неизменяемыми, подобно объектам класса `String` из языка Java.

В языке F# строка — это класс. Отдельные символы, представляемые в Unicode UTF-16, могут быть доступны, но не изменены. Строки могут подвергаться конкатенации с помощью операции +. В языке ML строка принадлежит к элементарному неизменяемому типу. Здесь предлагается операция конкатенации, обозначаемая значком (^), а также функции для выделения подстроки и определения размера строки.

Языки Perl, JavaScript, Ruby и PHP содержат встроенные операции сопоставления с образцом. Форма записи образца очень близка к математической нотации регулярных выражений. Регулярные выражения прямо используются в операциях сопоставления с образцом языка Perl.

Возможности сопоставления с образцом (на основе регулярных выражений) представляются средствами библиотек классов для языков C++, Java, Python, C# и F#.

Записи

В общем случае объект данных может включать в себя переменные различных типов. Записи позволяют сгруппировать переменные, относящиеся к объекту, и рассматривать их как единый модуль.

Запись — это неоднородная структура данных, в которой отдельные элементы идентифицируются именами.

В языке Pascal тип записи с k полями может представляться в виде

```
type
record
    <имя1>:  <тип1>;
    <имя2>:  <тип2>;
    . . . . . .
    <имяk>:  <типk>;
end
```

где <имя$_i$> — имя i-го поля, <тип$_i$> — тип i-го поля.

Каждое поле внутри записи имеет собственное имя.

Пример. Запись для комплексного числа может быть представлена так:

```
type complex =   record
    re:  real;
    im:  real;
end;
```

Объявления полей одинакового типа могут быть объединены:

```
type complex =   record
    re, im:  real;
end;
```

На языке Ada подобный тип записывается в виде:

```
type complex is
    record
        re, im:  float;
    end record;
```

Изменение порядка полей в записи не меняет смысла программы, так как доступ к полям выполняется по имени, а не по относительной позиции (как в массиве).

Объявление типа запись считается просто шаблоном. Память под запись выделяется при применении шаблона для объявления переменной:

```
var x, y, z : complex;
```

С переменными `x`, `y`, `z` связана память, размещение в этой памяти определяется типом `complex`.

Если выражение `E` обозначает запись с полем `f`, то само поле обозначается `E.f`.

Выражение `E.f` имеет значение и место расположения.

В `z.re := x.re + y.re` сумма значений полей `x.re` и `y.re` помещается в место расположения `z.re`.

В языках Pascal, Ada разрешены присваивания записей целиком:

```
x := y;
```

Данное присваивание устанавливает для `x.re` значение `y.re`, а для `x.im` — значение `y.im`.

Заметим, что в языках Pascal и Ada объекты-записи не могут быть анонимными, для них предварительно должен объявляться тип с именем.

В языке C типу запись соответствуют структуры:

```
struct <имя> {
    <тип1>  <имя_перем1>;
    …
    <типn>  <имя_перемn>;
};
```

Доступ к элементам структур обеспечивается с помощью:

- операции-точки `<имя_структуры>.<имя_элемента>`;
- операции-стрелки `<указатель_структуры> -> <имя_элемента>`.

Пример. Опишем структуру с именем `block` и заполним ее поля значениями: одно поле с помощью операции-точки, а другое поле — с помощью операции-стрелки.

```
struct block {
    int x;
    float y;
};
int main ()
{
    struct block a, *ptr_a = &a;
    a.x = 5;
    ptr_a -> y = 4.5;
}
```

Операция-стрелка ориентирована на использование указателя. Чтобы указатель `ptr_a` содержал адрес переменной `a` типа «структура», указатель инициируют. Забегая вперед, отметим, что адрес заносится с помощью операции взятия адреса `&a`.

В языке C имена чувствительны к регистру. Для того чтобы избавиться от влияния регистра на смысл имени, можно воспользоваться объявлением синонимов:

```
typedef struct block Block;
```

Теперь имена `block` и `Block` соответствуют одной и той же структуре и можно применить, например, следующее объявление:

```
Block b;
```

В языках C++ и C# записи также поддерживаются с помощью типа `struct`. В C# `struct` является типом, экземпляры которого размещаются в стеке, в отличие от объектов классов, которые размещаются в куче. Тип `struct` в C++ и C# обычно используется только для инкапсуляции данных, которую мы будем обсуждать в главе 15. Структуры также включены в ML и F#.

В языках Python и Ruby записи могут быть реализованы как хеши, которые сами по себе могут быть элементами массивов.

В языках Java и C# записи могут определяться как элементы данных в классах, с вложенными записями, которые рассматриваются как вложенные классы. Элементы данных в таких классах служат полями записи.

Язык Lua позволяет использовать в качестве записей ассоциативные массивы. Например, рассмотрим следующее объявление:

```
professor.name = "Ivanov"
professor.salary = 5000
```

Эти операторы присваивания создают таблицу (запись) по имени `professor` с двумя инициализированными полями `name` и `salary`.

Атрибуты объекта данных типа запись видны из приведенных выше объявлений:

1) количество полей;
2) тип данных для каждого поля;
3) имя для обозначения каждого поля.

Выбор полей является одной из основных операций над записями, например `a.x`. Она соответствует выбору элементов из массива при помощи индексов, но с одним отличием: индекс здесь всегда является буквальным именем поля и никогда не может быть вычисляемым значением. Приведенный пример выбора первого поля записи, `a.x`, соответствует выбору первого элемента массива, `M[1]`, но для записей не существует операции, аналогичной операции выбора из массива `M[X]`, где `X` — вычисляемое значение.

Операции над записью как единым целым обычно немногочисленны. Чаще всего используется операция присваивания одной записи некоторой другой, имеющей такую же структуру, например:

```
struct block c;
…
a = c;
```

где `c` — запись, имеющая те же атрибуты, что и `a`.

В памяти запись может храниться в единой области, в которой поля расположены последовательно. Для указания типов данных или других атрибутов отдельных полей могут потребоваться дескрипторы, но обычно для полей записи не требуется никаких дескрипторов во время выполнения программы.

Поскольку индексы отдельных полей (имена полей) известны еще во время компиляции (их не надо вычислять в период выполнения программы), то операция выбора реализуется достаточно просто. Ведь объявление записи позволяет узнать размер каждого поля и его позицию внутри области памяти еще во время компиляции. Таким образом, смещение каждого поля также можно определить во время компиляции. Формула вычисления адреса *i*-го поля записи `R` имеет вид:

$$A(R.i) = l\text{-значение } (R.i) = base + \sum_{j=1}^{i-1} (\text{размер } R.j),$$

где `base` — базовый адрес области памяти, содержащей запись `R`, а `R.j` — поле под номером `j`. Суммирование требуется из-за возможного различия размеров каждого поля. Но эту сумму всегда можно определить заранее, во время компиляции, и тем самым получить значение смещения S_i для i-го поля, так что во время выполнения потребуется добавить только базовый адрес области памяти:

$$A(R.i) = l\text{-значение } (R.i) = base + S_i.$$

Известно, что отводимые для хранения некоторых типов данных области памяти должны начинаться с определенных адресов. Например, область памяти, выделяемая целому числу, должна находиться на границе слова. Если же в компьютере реализована адресация каждого байта, то это значит, что адрес области памяти под целое число должен быть кратен четырем (двоичный код адреса должен заканчиваться двумя нулями). Следовательно, отдельные элементы записей не всегда могут располагаться слитно. Например, для структуры языка C, определяемой объявлением

```
struct ПреподавательКафедры
{    char Кафедра;
     int  Идентификатор;
} Преподаватель;
```

область памяти, отводимая под поле `Идентификатор`, должна начинаться на границе слова, и поэтому три байта между полями `Кафедра` и `Идентификатор` никак не используются и не содержат полезной информации. Фактически хранимая в памяти запись соответствует другому объявлению:

```
struct ПреподавательКафедры
{    char Кафедра;
     char НеиспользуемыйБлок[3];
     int  Идентификатор;
} Преподаватель;
```

Возможны два варианта реализации операции присваивания целой записи некоторой другой, обладающей такой же структурой:

- простое копирование содержимого области памяти, хранящей первую запись, в область памяти, предназначенную для второй записи;
- последовательность операций присваивания полям второй записи значений отдельных полей первой записи.

Записи и массивы со вложенными структурами

В языках программирования, где предусмотрены и записи, и массивы, обычно допускается их взаимное вложение. Например, может оказаться полезным такой объект данных, как массив, элементами которого являются записи. Например, следующее объявление языка C

```
struct ТипПреподавателя
{   int Ид;
    int Возраст;
    float Зарплата;
    char  Кафедра;
} Преподаватель [250];
```

описывает массив, состоящий из 250 элементов, каждый из которых является записью типа `ТипПреподавателя`. Доступ к элементу такой сложной структуры осуществляется с помощью двух операций выбора: сначала выбирается элемент массива, а затем — поле записи, например `Преподаватель [15].Зарплата`.

Запись также может состоять из полей, которые являются массивами или другими записями. В результате можно создавать записи, имеющие иерархическую структуру, на верхнем уровне которой располагаются поля, которые сами являются массивами или записями. Элементами второго уровня иерархии также могут быть записи или массивы.

Механизм размещения в памяти массивов и записей, элементами которых являются другие массивы и записи, считается простым расширением механизма размещения простых массивов и записей. Массив, состоящий из записей, размещается в памяти точно так же, как и простой массив, составленный из целых чисел или элементов другого элементарного типа. Отличие лишь в одном: область памяти под элемент массива (в области большего размера под сам массив) выделена для записи. Следовательно, массив записей размещается в памяти примерно так же, как и массив, состоящий из других массивов, но каждая строка в нем обеспечивает сохранение записи. Аналогично запись, полями которой являются записи (или массивы), сохраняет свою последовательную структуру размещения в памяти, но подобласти, отведенные под отдельные поля, обеспечивают представления целых записей.

Сравнение массивов и записей

1. Массив — это однородная коллекция элементов, все его элементы имеют одинаковый тип. То есть тип элемента `A`[i] известен в период компиляции, хотя действительный элемент, который обозначается как `A`[i], зависит от значения i, определяемого в период выполнения.
2. Запись — это неоднородная коллекция элементов, каждый ее элемент имеет свой собственный тип. Компоненты записи выбираются по именам, которые известны в период компиляции. Типы выбранных элементов также известны в период компиляции. Имя `f` в выражении `E.f` однозначно определяет поле, а тип `E.f` является типом поля с именем `f`.
3. Компоновка массива выполняется в период компиляции, а размещение элементов массива в памяти — в период выполнения программы.
4. Компоновка записи и размещение ее элементов в памяти выполняется в период компиляции.
5. Так как массив является однородной коллекцией, то каждый элемент занимает одинаковую область памяти. Если каждый элемент занимает `w` единиц, то память

для A[2] начинается после w единиц для A[1], а память для A[3] — через 2 × w единиц от начала и т. д.

6. Так как запись является неоднородной коллекцией, то ее элементы могут иметь разные типы и занимать области памяти разного размера. Имя поля известно в период компиляции, но память под поле может быть где угодно. Не требуется, чтобы поля записи находились в смежных ячейках (хотя при этом упрощается размещение и удаление памяти).

Объединения и вариантные записи

Обычные записи представляют объекты данных с общими атрибутами (одинаковым перечнем полей). *Вариантные записи* представляют объекты, в которых лишь часть атрибутов (полей) является общей.

Объединение — это специальный случай вариантной записи с пустой общей частью.

Вариантные записи имеют часть, общую для всех записей этого типа, и вариантную часть, специфичную для некоторого подмножества записей.

Суть вариантных записей рассмотрим на примере. Рассмотрим дерево выражения (рис. 10.5).

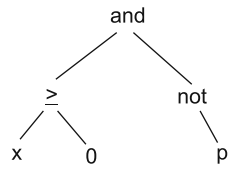


Рис. 10.5. Дерево выражения

В этом дереве присутствуют узлы следующих категорий: переменные, константы, двухместные операции и одноместные операции.

Положим, что все узлы имеют по два общих атрибута (имя, категория), а остальные атрибуты у них разные. Например, они могут иметь разное количество потомков. Узлы констант и переменных не имеют потомков. Узлы для двухместных операций имеют по два потомка, а узлы для одноместных операций — одного потомка.

Все эти узлы можно представить как вариантные записи, структуру которых иллюстрирует рис. 10.6.

Как видим, в первой категории вариантная часть пуста, во второй категории она содержит одно поле потомок, а в третьей категории — два поля: л_потомок и п_потомок. Поле тега K здесь используется для задания различий между вариантами.

В языке Pascal объявления вариантных частей записываются в виде:

```
Case  <имя_тега>:  < имя_типа> of
    <constant1>:  (<поля1>);
    <constant2>:  (<поля2>);
    . . . . . .
    <constantm>:  (<поляm>);
```

Константы соответствуют состояниям вариантной части; каждое состояние имеет свой собственный набор полей. Выбираемое состояние задается значением поля тега (с именем <имя_тега> и типом < имя_типа>).

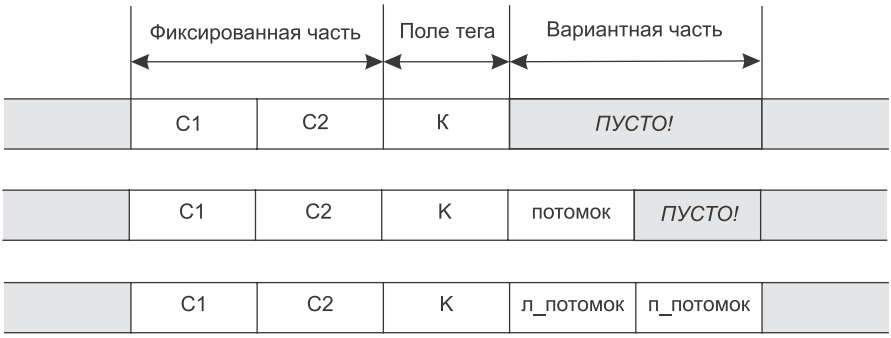


Рис. 10.6. Структура узлов для дерева выражения

Представим вариантный тип записи `узел` с тремя состояниями, моделирующий структуру на рис. 10.6, предварительно объявив вспомогательный тип `вид`:

```
type   вид = (лист,  один,  два);
    узел = record
        C1 : T1;
        C2 : T2;
        Case K : вид  of
            лист : ( );
            один : (потомок : T3);
            два : (л_потомок,  п_потомок : T4);
        end;
```

Далее можно объявить объект данных этого типа:

```
Var вершина : узел (два);
```

В объявлении указан не только тип объекта `узел`, но и значение тега, задающее настройку на конкретный вариант. Вариантная часть типа `узел` имеет наборы полей, соответствующие константам: `лист`, `один`, `два`. Выбор между этими тремя вариантами зависит от значения тегового поля `K`.

В этой записи всегда присутствуют поля `C1` и `C2`. Если значение тега `K` равно `лист`, то в записи больше полей нет, в то время как если `K = два`, то в записи дополнительно будут содержаться поля `л_потомок` и `п_потомок`.

Выбор поля вариантной записи аналогичен выбору поля обычной записи. Например, `вершина.C1` и `вершина.п_потомок` обозначают поля из определенного ранее варианта объекта `вершина`, имеющего тип `узел`. Для обычных записей каждое поле существует в течение всего времени жизни записи, а в случае вариантной записи поле может просуществовать какое-то время (пока тег имеет соответствующее значение), затем прекратить свое существование (если тег меняет значение) и вновь появиться при установке первоначального значения тега. Следовательно, имя `вершина.п_потомок` может указывать поле, которое в данный момент не существует. Проблема адресации несуществующего поля вариантной записи может иметь следующие решения:

- *Динамическая проверка.* В ходе вычислений обращение к нужному полю можно предварять проверкой значения тега. Проверка позволит судить, существует ли поле в данный момент. Если значение тега правильное, то нужное поле доступно; в противном случае фиксируется ошибка периода выполнения.

- *Отсутствие проверки*. Очень часто допускается объявление вариантной записи без явного указания тега. Однако если запрашиваемое поле отсутствует, то существующие поля вариантной записи могут быть некорректно использованы или повреждены. Язык Pascal разрешает создание вариантных записей без определения тегов, а в объединениях `union` языков C и C++, как мы увидим чуть позже, тег вообще отсутствует.

При размещении вариантной записи в памяти всегда страхуются. В ходе компиляции вычисляют размер памяти, необходимой для хранения всех полей каждого варианта. Поскольку варианты альтернативны друг другу и не могут существовать одновременно, в целом под запись выделяется область памяти, достаточная для размещения самого емкого (большого) варианта. Внутри этой области вычисляются смещения полей для каждого возможного варианта (выполняется так называемая «компоновка варианта»). Используется информация о количестве и типах полей. Смещения применяются при адресации конкретных полей на этапе вычислений: конкретное смещение складывается с базовым адресом записи. Каждый вариант предусматривает использование своего собственного набора смещений. Конечно, при подобном подходе часть памяти в вариантной записи может пустовать, зато исключается возможность нехватки памяти. При применении вариантной записи самой главной является проблема *актуальности* поля: не ищем ли мы то, чего уже (или еще) нет? Как уже упоминалось, ответ на этот вопрос дает динамическая проверка тега.

В языках C и C++ вариантные записи отсутствуют, здесь предусмотрена лишь крайне упрощенная версия — объединение (`union`), в которой нет никакой проверки типа при использовании. По этой причине объединения в этих языках называют свободными, подчеркивая тот факт, что в данном случае программист совершенно освобождается от контроля типов.

Например, рассмотрим следующее объединение на языке C:

```
union free {
    int a;
    float b;
};
union free y;
float x;
. . .
y.a = 45;
x = y.b;
```

В этом примере объявлена переменная `y` с типом объединения по имени `free`, а также вещественная переменная `x`. В какой-то момент времени в целое поле `a` переменной `y` занесено значение. В последнем операторе присваивания тип не проверяется, поскольку система не способна определить текущий тип текущего значения переменной `y`; следовательно, переменной `x` присваивается битовая строка `45`, что, конечно, нелепо.

Во многих языках объединения — это потенциально небезопасные конструкции. Проверка типа требует наличия в объединении индикатора типа. Такой индикатор называется тегом, или дискриминантом, а объединение с дискриминантом называют *дискриминантным объединением*. Первым языком, поддерживающим дискриминантные объединения, был Algol 68. Сейчас они применяются в языках ML, Haskell и F#.

В языках Java и C# объединения отсутствуют, что отражает требования к повышению безопасности в языках программирования.

Вариантные записи ослабляют надежность типов?

Компиляторы обычно не проверяют соответствие значения тега состоянию записи. Кроме того, теговые поля не обязательны.

В языке Pascal проверка типов вариантных записей считается невозможной. Выдвигают две причины:

- Программа может изменять тег без изменения какого-либо варианта, поэтому разработчики языка игнорируют возможность проверки.
- Программист может пропустить тег, делая запись свободным объединением.

Имя тега вообще может быть пропущено, как в следующем объявлении типа `t`:

```
type   вид =  1 .. 2;
t = record
    Case  вид  of
        1:  (i: integer);
        2:  (r: real);
end;
```

Тип записи `t` состоит только из вариантной части, то есть является объединением. По замыслу автора, имя типа `вид` после слова `Case` обеспечивает, что вариантная часть может быть в одном из двух состояний.

Но здесь нет имени тега, поэтому система полагает, что нет и тегового поля. Если есть объявление

```
var  x : t;
```

то возможные поля для переменной: `x.i` или `x.r`.

Состояние в записи не запоминается, поэтому нельзя проверить, в каком реальном состоянии находится объект `x`.

В языке Ada синтаксис и семантика вариантных записей языка Pascal были значительно улучшены, обеспечив существенное повышение безопасности. Здесь решены обе проблемы вариантных записей языка Pascal:

- запрещено изменение тега без изменения варианта;
- тег обязателен во всех вариантных записях.

Мало того, в языке Ada тег проверяется при любых обращениях к вариантным объектам. Перепишем рассмотренный выше вариантный тип `узел` в терминах языка Ada:

```
type вид is (лист, один, два);
type узел (K : вид) is
    record
        C1 : string (1..4);
        C2 : integer;
        case K is
            when лист ⇒
                null;
            when один ⇒
                потомок : string (1..4);
```

продолжение ⇩

```
            when два ⇒
                л_потомок,  п_потомок : string (1..4);
        end case;
    end record;
```

Здесь полная информация о теге записывается в заголовке типа (в круглых скобках) и называется дискриминантом типа. При объявлении переменной значение дискриминанта нужно обязательно указывать:

```
вершина : узел (два);
```

Переменная `вершина` называется *ограниченной вариантной переменной*. Ее вариант выбран и не может меняться при выполнении программы. Тег, использованный при объявлении ограниченной вариантной переменной, интерпретируется как именованная константа.

Существует и другая возможность. Если в объявлении вариантного типа предусмотрено значение по умолчанию, то на его основе можно объявлять *неограниченные вариантные переменные*. Например, на основе типа

```
type узел (К : вид := лист) is
    record
        С1 : string (1..4);
        С2 : integer;
        case К is
            when лист ⇒
                null;
            when один ⇒
                потомок : string (1..4);
            when два ⇒
                л_потомок,  п_потомок : string (1..4);
        end case;
    end record;
```

можно объявить неограниченную вариантную переменную:

```
вершина2 : узел;
```

При ее объявлении тег не указывается, вариант переменной выбирается по умолчанию (в данном случае это `лист`), но появляется дополнительная возможность: изменение варианта переменной во время выполнения программы. Однако вариант переменной может поменяться только при присвоении ей (например, с помощью агрегата) значения целой записи, в том числе и тега. Например:

```
вершина2 := (К ⇒ один, С1 ⇒ "not ", С2 ⇒ 2, потомок ⇒ "extn");
```

Подобное решение полностью исключает возможность несовместимости вариантных записей:

- если присваивается набор констант, то значение тега и вида варианта подвергается статической проверке на совместимость;
- если присваивается значение переменной, то совместимость гарантируется самим процессом присваивания, поскольку переменная из левой части становится копией переменной из правой части.

Таким образом, эта реализация вариантных записей совершенно безопасна и обеспечивает проверку типов во всех случаях, хотя неограниченные вариантные переменные и должны проверяться динамически.

Множества

Множество — это структура, содержащая неупорядоченный набор *различных* значений. В Паскале значения множеств задаются записью элементов в квадратных скобках:

```
['0' .. '9']    ['a'.. 'z', 'A'.. 'Z']          [ ]
```

Все элементы множества должны быть одного и того же простого типа — целые числа, перечисления или поддиапазоны в этих типах.

Объявление множественного типа

```
type <ТипМножества> = set of <БазовыйТип>;
```

определяет все подмножества, составленные из комбинаций значений базового типа. Если базовый тип имеет n значений, то тип множества позволяет генерировать 2^n значений.

Для переменной `A`

```
var A : set of [1 .. 3];
```

разрешены подмножества:

```
[ ],  [1],  [2],  [3],  [1,2],  [1,3],  [2,3],  [1,2,3]
```

Все эти множества представляются с помощью трех битов. Элемент `1` представляется первым битом, элемент `2` — вторым битом, а элемент `3` — третьим битом. Множество `[1,3]` кодируется битовым вектором `101`.

Множество из n элементов реализуется как битовый вектор длины n. Длина этого вектора обычно кратна длине машинного слова. В языке Pascal количество элементов во множестве должно находиться в диапазоне `0..255`.

Перечень операций, предусмотренных для множественного типа, приведен в табл. 10.3.

Таблица 10.3. Набор операций для типа «множество» в языке Pascal

Операции	Пояснения
x in A	Проверка принадлежности
A + B	Объединение $A \cup B$
A – B	Разность $A - B = \{x \mid x \in A \text{ and } x \notin B\}$
A * B	Пересечение $A \cap B$
A / B	Симметричная разность $(A - B) \cup (B - A)$
$\leq$, = , $\neq$, $\geq$	Сравнения

Базовой операцией над множествами является проверка принадлежности. Операция `in` проверяет, принадлежит ли элемент `x` множеству `A`, то есть дает ответ на вопрос: верно ли, что $x \in A$?

Битовые векторы обеспечивают эффективную реализацию типовых операций над множествами (на основе битовых операций), как то: *объединение*, *разность* и *пересечение* множеств. Напомним их смысл. Пусть имеются два множества `A` и `B`. Каждая из этих операций создает новое множество `C`, которое либо содержит элементы обоих множеств `A` и `B` (в случае *объединения*), либо содержит только те элементы, которые принадлежат и множеству `A`, и множеству `B` (в случае *пересечения*), либо

содержит только те элементы A, которых нет в B (в случае *разности* множеств A и B), либо содержит все те элементы A, которых нет в B, а также те элементы B, которых нет в A (в случае *симметричной разности* множеств A и B).

Дополнительными операциями могут быть: вставка и уничтожение отдельных значений. Первая операция включает x во множество A, если там еще нет такого элемента. Вторая операция удаляет x из A, если x является элементом множества.

Множества могут сравниваться с помощью операций отношения ≤ , = , ≠ , ≥ , где ≤ интерпретируется как «первый элемент является подмножеством второго?», а ≥ интерпретируется как «первый элемент является надмножеством второго?» Операции < и > запрещены, поскольку в случае множеств не имеют смысла.

Заметим, что к элементам множества невозможно осуществить доступ при помощи индексов или по их взаимному расположению.

Множественный тип очень удобен при программировании сложных жизненных ситуаций, поскольку позволяет наглядно и компактно отобразить замысловатые логические решения (за это и приветствуют аппарат теории множеств).

Пример. Существует набор продуктов, продаваемых в нескольких магазинах. *Определить*: какие продукты есть во всех магазинах; полный набор продуктов в городе.

```
program  Shops;
    type Food = (hleb, moloko, mjaso, syr, sol, maslo);
         Shop = set of Food;
     var IKI, Rimi, Maxima, MinFood, MaxFood : Shop;
Begin
        IKI := [hleb, moloko]; Rimi := [ hleb, syr, maslo];
         …
        MinFood := IKI * Rimi * Maxima;
        MaxFood := IKI + Rimi + Maxima;
End.
```

Кортежи

Кортеж — это структура данных из разнотипных элементов, не имеющих имен и представляемых значениями. Кортежи схожи с записями, но отличаются от них тем, что в записях элементы (поля) имеют имена, а в кортеже нет.

Язык Python содержит неизменяемый тип кортеж (`tuple`). Если кортеж нужно изменить, то с помощью функции `list` его конвертируют в массив. После изменения он может быть преобразован обратно в кортеж функцией `tuple`.

Кортеж удобно использовать в качестве защищенного от записи параметра функции. Если параметром является кортеж, функция лишается возможности изменить его значение.

Кортежи в языке Python очень близки к его спискам, разница лишь в том, что кортежи — это неизменяемые структуры. Следующий пример показывает, что кортеж создается присвоением литералов:

```
aTuple = (7, 9.5, 'peace')
```

Как видим, все элементы разнотипны.

На элементы кортежа можно ссылаться по индексу, записываемому в квадратных скобках:

```
aTuple[1]
```

ПРИМЕЧАНИЕ

В языке Python строки записываются в апострофах.

Это ссылка на первый элемент кортежа, поскольку индексация в кортеже начинается с единицы.

Кортежи подвергаются конкатенации, обозначаемой операцией плюс (+). Они могут удаляться оператором `del`. Над кортежами определены и другие операции и функции.

Язык ML также поддерживает тип данных «кортеж». Кортеж в ML должен иметь по меньшей мере два элемента, в то время как в Python кортежи могут быть пустыми или содержать один элемент. Подобно языку Python, кортеж в ML может включать в себя элементы разных типов. Следующий оператор создает кортеж:

```
val aTuple = (7, 9.5, 'peace');
```

Принят следующий синтаксис доступа к элементу кортежа:

```
#1(aTuple);
```

Это ссылка на первый элемент кортежа.

Новый тип кортежа определяется в ML следующим объявлением типа:

```
type intReal = int * real;
```

Значения этого типа состоят из целых и вещественных чисел.

В языке F# также предусмотрены кортежи. Для создания кортежа используется оператор `let`. В нем имя кортежа указывается в левой части присваивания, а значение записывается в правой части как список выражений, отделяемых друг от друга запятыми. Этот список помещается в круглые скобки. Если в кортеже два элемента, на них можно ссылаться с помощью функций `fst` и `snd` соответственно. При большем числе элементов применяют паттерн кортежа, размещаемый в левой части оператора `let`. Паттерн кортежа — это простая последовательность имен, по одному имени на каждый элемент кортежа. Паттерн может ограничиваться круглыми скобками или обходиться без них. Присутствие паттерна кортежа в левой части конструкции `let` свидетельствует о множественном присваивании. В качестве примера рассмотрим следующие конструкции `let`:

```
let tup = (4, 7, 9);;
let x, y, z = tup;;
```

Здесь имени `x` присваивается значение 4, имени `y` — значение 7, а имени `z` — значение 9.

В языках Python, ML и F# кортежи позволяют функциям возвращать множественные значения.

Списки

Список — это упорядоченная последовательность некоторых структур данных. Первый элемент списка называют *головой*, а остальные элементы — *хвостом* списка. Как правило, типы элементов списка явно не объявляются. Применяют как однородные, так и неоднородные списки. Списки редко имеют фиксированную длину, обычно их длина увеличивается и уменьшается в период выполнения программы.

Впервые списки появились в функциональном языке LISP. Они всегда были неотъемлемой частью функциональных языков, но в последние годы стали все чаще использоваться и в императивных языках.

В языках Scheme и Common LISP списки записываются внутри скобок в виде элементов, которые не отделяются друг от друга знаками пунктуации. Например,

```
(a b c d)
```

Вложенные списки имеют ту же форму:

```
(a (b c) d)
```

В этом списке внутренний список `(b c)` вложен во внешний список.

В языке LISP (и его последователях) данные и программный код записываются одинаково. Если список `(a b c)` интерпретировать как программный код, то он означает вызов функции `a` с параметрами `b` и `c`.

В языке Scheme фундаментальными операциями над списками являются две функции:

- функция выбора части списка;
- функция создания списка.

Функция `car` возвращает первый элемент списка, являющегося ее параметром. Рассмотрим следующий пример:

```
(car '(a b c))
```

Апостроф перед параметром-списком указывает, что интерпретатор не должен рассматривать список как вызов функции `a` с параметрами `b` и `c`. При такой записи функция `car` просто возвращает `a`.

Функция `cdr` возвращает свой параметр-список без первого элемента:

```
(cdr '(a b c))
```

Результатом этого вызова функции является список `(b c)`.

Язык Common LISP дополнительно предлагает функции `first` (аналог `car`), `second`, . . . , `tenth`, возвращающие тот элемент из списка-параметра, который определен именем функции.

Новые списки в языках Scheme и Common LISP создаются функциями `cons` и `list`. Функция `cons` принимает два параметра и возвращает новый список, в котором первым элементом будет первый параметр. Продолжение этого списка задается вторым параметром функции. Например:

```
(cons 'a '(b c))
```

Данный вызов вернет новый список `(a b c)`.

Функция `list` принимает любое количество параметров и возвращает новый список, составленный из этих параметров. Рассмотрим следующий вызов функции `list`:

```
(list 'a 'b '(c d))
```

Его результатом становится новый список `(a b (c d))`.

Язык ML также определяет списки и операции над списками, но их внешний вид отличается от того, который используется, например, в языке Scheme. Списки записываются в квадратных скобках, а элементы отделяются друг от друга запятыми:

```
[5, 7, 9]
```

Запись `[]` обозначает пустой список, которому соответствует пометка `nil`.

Функция языка Scheme с именем `cons` реализована в ML как бинарная инфиксная операция и обозначается значком «двойное двоеточие» (::). Например, операция

```
5 :: [6, 8, 9]
```

возвращает следующий новый список: `[5, 6, 8, 9]`.

В отличие от LISP элементы списка в ML должны иметь одинаковый тип, поэтому следующий список некорректен:

```
[3, 5.8, 7]
```

Кроме того, в языке ML существуют функции `hd` (head) и `tl` (tail), которые соответствуют функциям `car` и `cdr` языка Scheme. Например,

- вызов `hd [5, 7, 9]` возвращает 5.
- вызов `tl [5, 7, 9]` возвращает `[7, 9]`.

Списки в F# похожи на списки ML, но имеют синтаксические особенности. Так, элементы списка здесь отделяются точкой с запятой. Операции `hd` и `tl` те же самые, но они называются методами класса `List`, поэтому возможен следующий вызов

```
List.hd [1; 3; 5; 7],
```

который в данном случае вернет 1.

Операция `cons` в F# обозначается двумя двоеточиями, как в ML.

В языке Python предусмотрен тип данных «список», который используется так же в массивах этого языка. В отличие от Scheme, Common LISP, ML и F#, списки в Python считаются изменяемыми. Они могут содержать любое значение данных или объект. Список в языке создается присваиванием имени списка значений. Список значений помещается в квадратные скобки и представляет собой последовательность выражений, отделяемых запятыми. Например:

```
aList = [2, 7.4, "book"]
```

На элементы списка ссылаются с помощью индексов в квадратных скобках:

```
y = aList[2]
```

Этот оператор присваивает значение `"book"` переменной `y` (индексация элементов начинается с нуля). Список элементов может быть обновлен присваиванием. Элемент списка можно удалить по оператору `del`:

```
del aList[2]
```

Этот оператор удаляет третий элемент списка `aList`.

В Python предусмотрен достаточно мощный механизм для создания списков, называемый генератором списков. Генератор списков следует традиции создания множеств; он позволяет создавать новые списки, выполняя выражение для каждого элемента в последовательности, по одному за раз, двигаясь слева направо. Генераторы списков заключены в квадратные скобки (чтобы отразить тот факт, что они создают список) и составлены из выражения и конструкции цикла, которые используют одно и то же имя переменной.

Синтаксис генератора списков имеет следующий вид:

```
[выражение for имя_переменной in список if условие]
```

Рассмотрим пример:

```
[a * a for a in range(12) if a % 3 = = 0]
```

Функция `range` создает список [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Условие отфильтровывает все числа в списке, отбрасывая все числа, которые не делятся на 3. Далее выражение возводит каждое оставшееся число в квадрат. Квадраты чисел накапливаются в возвращаемом списке. Данный генератор списков возвращает следующий список:

```
[0, 9, 36, 81]
```

ПРИМЕЧАНИЕ

Функция `range` с одним аргументом генерирует список целых чисел в диапазоне от нуля до указанного в аргументе значения, не включая его. Если функции передать два аргумента, первый будет рассматриваться как нижняя граница диапазона. Необязательный третий аргумент определяет шаг — в этом случае интерпретатор будет добавлять величину шага при вычислении каждого последующего значения (по умолчанию шаг равен 1).

Сечения списков также поддерживаются в языке Python.

Генераторы списков в языке Haskell представляются в следующей форме:

```
[тело | квалификаторы]
```

Например, рассмотрим следующее определение списка:

```
[n * n | n <- [1..10]]
```

Здесь задается список квадратов чисел в диапазоне от 1 до 10.

Язык F# также содержит генераторы списков, которые могут использоваться для создания списков (массивов). В качестве примера рассмотрим следующий оператор:

```
let myList = [|for i in 1 .. 5 -> (i * i) |];;
```

Этот оператор создает список `[1; 4; 9; 16; 25]` с именем `myList`.

Языки C# и Java поддерживают коллекции родовых динамических классов, `List` и `ArrayList` соответственно. По сути, эти структуры являются списками.

Контрольные вопросы и упражнения

1. Предположим, что объявление массива `M` задано с использованием некоторого перечисляемого типа в качестве набора индексов. Например:

   ```
   ExamsMark = (Math, Programming, Prog_Languages, Computers, Philosophy);
   M: array [ExamsMark] of real;
   ```

   - Предложите подходящий способ представления в памяти массива `M` (включая дескриптор) и выведите адресную формулу доступа для вычисления местоположения элемента `M[`i`]`.
   - Покажите, как изменятся способ представления и формула доступа, если `M` будет объявлен следующим образом:

     ```
     M: array [Prog_Languages .. Philosophy] of real;
     ```

2. Рассмотрим представление множества через битовые строки. Предположим, что максимальное количество элементов в множестве должно быть меньше, чем

длина слова, обусловленная аппаратурой компьютера. Предложите алгоритмы реализации операций объединения, пересечения и разности множеств и определения принадлежности к множеству (операция `in`) на основе следующих встроенных в аппаратуру простейших операций: `логическое И`, `логическое ИЛИ` и `логическое дополнение`, применяемые к целым словам

3. Для языка, в котором допускаются вариантные записи без полей тегов (свободное объединение), как, например, в языке Pascal, напишите процедуру

```
procedure Example (i: integer; var r: real;)
```

которая использует запись с двумя вариантами. Единственным назначением этой процедуры является попытка сбить с толку систему проверки типов. Процедура `Example` получает параметр *i* целочисленного типа и возвращает в качестве результата ту же комбинацию битов, но как вещественное число `r`, при этом не выполняя фактического преобразования целого значения *i* к вещественному числу.

4. Положим, что имеется версия языка Pascal, в которой имена полей в записях (не вариантных) могут быть целыми числами, а при выборке полей разрешается вычислять целые значения имен полей, так что, например, для выборки компонента записи можно использовать выражение `r.(i + 2)`.
   - Объясните, почему в такой версии представление записей в памяти, описанное в разделе «Записи», перестанет быть адекватным.
   - Модифицируйте это представление так, чтобы оно могло функционировать в новой версии, и предложите адресную формулу доступа (или алгоритм), который можно было бы использовать в этом новом представлении для выбора поля `r.k`, где `k` — это вычисленное значение.
5. Повторите упражнение 4, но для записей с вариантными полями.

Глава 11
Указатели

Неотъемлемой частью языков программирования являются типизированные средства косвенного доступа к объектам данных — указатели. В этой главе обсуждаются основные характеристики, понятия указателей, операции над указателями; поясняется специфика построения на их основе динамических связных структур; рассказывается о сокращении затрат времени при работе (посредством указателей) со сложными структурами. Исследуются такие проблемы их применения, как повисшие указатели и утечки памяти. Анализируются, сравниваются безопасные и гибкие аппараты указателей, реализованные в различных языках. Описывается особая разновидность указателя — ссылка, введенная в языке C++, поясняются преимущества использования ссылок. В заключение разъясняются вопросы реализации указателей.

Основные понятия

Указатель — это величина, предоставляющая косвенный доступ к элементам известного типа. Указатели являются прообразами косвенных адресов в машинных командах. Главное отличие — указатель `p` указывает только на объекты определенного типа `T` [12, 17].

Указатели обеспечивают:

- *Повышение эффективности*. Вместо копирования или пересылки в памяти большой структуры данных можно скопировать или переслать только указатель на эту структуру.
- *Динамические структуры*. С помощью записей и указателей можно реализовать структуры данных, которые растут и сжимаются в период выполнения программы.

В языке Pascal указательный тип записывается в форме

```
type  <Имя_указательного_типа>  = ↑<Имя_типа>;
```

Например, объявление типа `link`, который является указателем на тип `cell`, имеет вид:

```
type link = ↑cell;
```

Все указательные типы `↑T` имеют одинаковый формат размещения, вне зависимости от типа `T`, на который они указывают. Объяснение достаточно простое:

значениями любого указательного типа являются адреса, а адреса, в общем случае, не зависят от типа указываемого объекта. Поэтому формат указателя известен, даже если тип T еще не объявлен.

ПРИМЕЧАНИЕ

Размер (формат) адреса зависит только от величины адресного пространства компьютера и специфики его организации.

Операции над указателями в языке Pascal

Базовая операция — это разыменование, то есть переход от указателя к значению указываемого объекта. В языке Pascal разыменование обозначается как постфиксная стрелка (p↑).

ПРИМЕЧАНИЕ

В языке Pascal символ ↑ имеет двойное назначение: как префиксная операция создания указательного типа и как постфиксная операция разыменования. Тип ↑T используется для создания указателей на объекты типа T. Выражение p↑ обозначает объект, указываемый с помощью p.

В языке Pascal объект данных типа T, доступ к которому должен обеспечиваться через указатель, создается с помощью оператора new(p), где p — указатель на переменную типа T. Такой объект называют *динамическим*, поскольку создается он в период выполнения программы. Память под объект выделяется из *динамической* области памяти по имени heap (куча). Объект существует до тех пор, пока не произойдет явное освобождение памяти по оператору dispose (p).

Специальное значение указателя nil соответствует случаю, когда указатель ни на что не указывает. Это значение может быть присвоено переменной любого указательного типа.

В языке Pascal возможны следующие операции над указателями [92, 93]:

1. *Динамическое размещение* в куче (heap). При выполнении new(p) в p заносится адрес нового объекта типа T, созданного в куче.
2. *Разыменование*. Выражение p↑ обозначает *r*-значение объекта данных, указываемого с помощью p.
3. *Присваивание*. Разрешено между указателями одинакового типа p := q.
4. *Проверка эквивалентности*. Выражение эквивалентности p = q проверяет, содержат ли два указателя одного типа адрес одного и того же объекта. Возможна проверка неэквивалентности.
5. *Освобождение*. Динамический объект данных существует до момента выполнения оператора dispose(p), по которому память из-под него освобождается.

Динамические связные структуры данных

Необходимость работы с растущими структурами данных характерна для программ типа компилятора, которые должны обрабатывать исходные тексты от нескольких строк до нескольких тысяч строк.

Растущие и сжимаемые (в ходе выполнения программы) структуры реализуются на основе записей и указателей. Их создание возможно благодаря соблюдению принципа статического формата — размер и формат памяти для каждого типа известны статически, перед выполнением программы.

Так как формат определяется в период компиляции, динамические структуры (растущие и сжимаемые в период выполнения) реализуются с помощью элемента фиксированного размера.

Связная структура данных растет, когда элемент размещается и связывается с другими элементами, структура данных сжимается, когда элемент удаляется.

Элементы и связи используются также для реализации таких структур данных, как связные списки и деревья. Элемент в простом связном списке имеет вид, представленный на рис. 11.1.

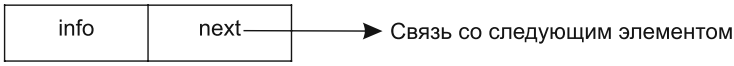


Рис. 11.1. Элемент связного списка

Как видим, у него две секции: в одной секции (по имени `info`) содержится порция данных, в другой — адрес соседнего элемента структуры.

Тип элемента может быть объявлен так:

```
type    link = ↑ cell;
    cell = record
        info : integer;
        next : link
    end;
```

Тип `link` используется для указания на `cell`, а тип `cell` — это запись с информационным полем и полем-указателем на следующий элемент. Заметим, тип `link` (указательный тип) объявлен перед объявлением типа `cell` и используется внутри объявления `cell`.

Пример. Допустим, что указательные переменные `front` и `p` имеют тип `link`. Создадим динамическую структуру данных вида, изображенного на рис. 11.2.

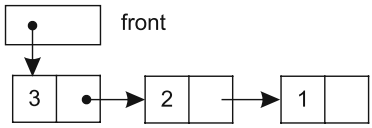


Рис. 11.2. Динамическая структура данных

Эта задача решается следующей последовательностью операторов:

```
1. front : = nil;  i : = 1;
2. new (p) ;              (*  поместить в p адрес нового элемента  *)
3. p↑. info : = i;        (* установить значение поля  info  элемента  *)
```

```
4. p↑. next  := front;   (*  установить значение поля  next  *)
5. front  := p;          (*  скопировать адрес во front  *)
```

Для добавления нового элемента надо выполнить `i+1` (готовимся к занесению в него новой информации) и снова перейти на вторую строку последовательности.

Логическая организация связной структуры не зависит от физического размещения ее элементов в памяти [94]. Нет необходимости, чтобы связанные элементы находились в смежных ячейках памяти. Они могут быть где угодно. При сохранении связей ячейки могут даже перемещаться, не разрушая структуру данных.

Повисшие указатели и утечки памяти

Работа с динамическими объектами таит в себе повышенную опасность. Рассмотрим источники этой опасности.

Повисший указатель — это указатель на память, которая используется для других целей. Типичный случай — память уже перераспределена. К повисшему указателю приводит оператор `dispose(p)`. Повисшие указатели понижают безопасность программ.

Память, которая распределена, но недоступна, называется *мусором* (*garbage*).

Говорят, что программы, которые создают мусор, дают *утечки памяти* (*memory leaks*). К утечкам памяти могут привести присваивания между указателями. Утечка памяти может понизить производительность, но не безопасность.

Предположим, что `p` и `q` — указатели на какие-то элементы (рис. 11.3).

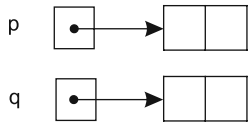


Рис. 11.3. Два указателя на объекты

Оператор присваивания `p := q` приводит к тому, что `p` и `q` указывают на один и тот же элемент (рис. 11.4).

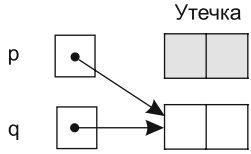


Рис. 11.4. Указатель p получает значение указателя q

Элемент, на который ранее указывал `p`, по-прежнему остается в памяти, но теперь он недоступен. Это утечка памяти.

Выполним теперь оператор `dispose(p)`, освобождающий память под элемент, на который указывал `p` (рис. 11.5).

Теперь оба указателя становятся «безработными», поскольку хранят адрес уничтоженного объекта. Хорошо это или плохо?

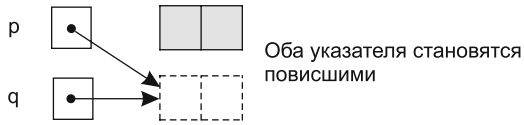


Рис. 11.5. Освобождение памяти и «повисание» указателей

Представьте себе, что в недалеком будущем эта ячейка памяти задействуется под новую, секретную информацию. Тогда безработные указатели смогут похитить все секреты!

Безопасность указателей в Паскале

Язык Паскаль проектировался в расчете на максимальную безопасность работы с этой особой категорией — *динамическими объектами* [92]. В чем их особенность? Любой динамический объект создается в период выполнения программы, собственного имени не имеет, доступен только косвенно, через указатель. Ну а указатель — источник повышенной опасности, так как результат разыменования повисшего указателя непредсказуем.

В этих условиях основой обеспечения безопасности считают разделение памяти на статическую и динамическую (рис. 11.6).

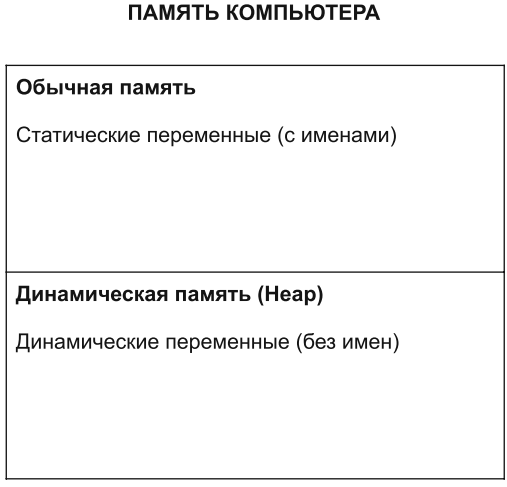


Рис. 11.6. Разделение памяти на статическую и динамическую

Динамическая память — это единственное жизненное пространство для указателей. Указатели могут ссылаться только на динамическую память. Указатели не могут использоваться для доступа к статической памяти под обычные переменные. Операции над указателями сохраняют разницу между статическими и динамическими переменными. Результаты операций над указателями никогда не приводят к пересечению границы между статической и динамической памятью. Иными словами, «граница на замке» и законопослушные статические переменные избавлены

от кошмаров повисших указателей. Да и мусор в статической памяти (по вине указателей) появиться не может.

Напомним, у операций с указателями два недостатка:

1. Утечки памяти (это неприятность).
2. Повисшие указатели (это серьезная опасность). Способ устранения — присваивать указателю (параметру оператора `dispose`) значение `nil`. Реализовать это можно как отдельный оператор или как завершающую фазу выполнения оператора `dispose`.

Указатели как посредники

Очень часто указатели используют для косвенного обращения к большим структурам данных, повышая скорость и сокращая затраты на организацию доступа.

Пример. Для работы с книжными индексами введем типы данных:

```
type  элемент = record
        термин :  предст_термина;
        страница :  integer
    end;
type  предст_термина = record
        данные :  array [0 .. 99] of char;
        длина :  integer
    end;
```

Для работы с элементами введем указательный тип

```
type  у_элемент = ↑элемент;
```

и объявим два указателя этого типа

```
var  e, f :  у_элемент; (* выделение памяти под указатели *)
```

Выделение памяти под структуры производится с применением оператора `new`:

```
new (e); (* выделение памяти под структуру *)
new (f); (* выделение памяти под структуру *)
…
```

Далее в программе каждая из структур наполняется информацией, содержание которой иллюстрируют рис. 11.7 и 11.8.

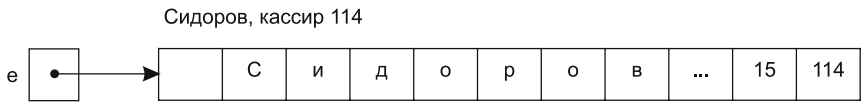


Рис. 11.7. Содержание структуры, адресуемой указателем e

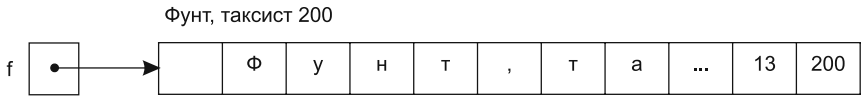


Рис. 11.8. Содержание структуры, адресуемой указателем f

Оценим эффективность созданной системы доступа к сложным структурам данных.

Перестановка указателей и перемещение данных

Перестановка указателей требует значительно меньше работы, чем перестановка записей, на которые они ссылаются.

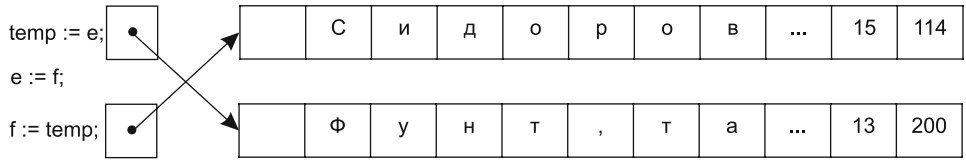


Рис. 11.9. Организация перекрестной адресации

На рис. 11.9 мы минимальными усилиями обеспечили перекрестную адресацию.

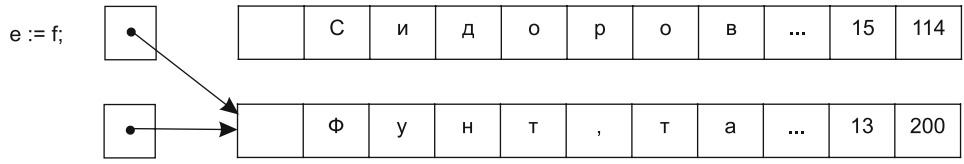


Рис. 11.10. Повышение надежности доступа к важной информации

На рис. 11.10 показано альтернативное решение: мы усилили надежность доступа к какой-то очень важной информации. Теперь эту запись обслуживают два указателя.

Все эти действия выполняются очень быстро.

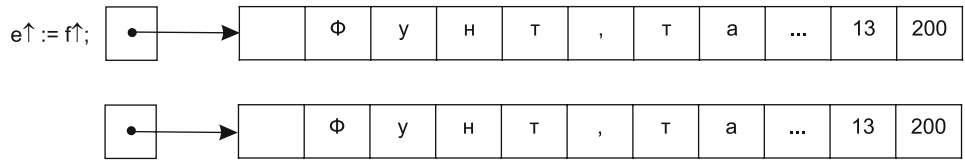


Рис. 11.11. Перемещение данных из одной области в другую область памяти

А вот на действие, которое иллюстрирует рис. 11.11, понадобится много времени. Объяснение простое: через механизм разыменования здесь инициирован процесс действительной пересылки большого количества элементов данных.

Указатели в языке Ada

Указатели в языке Ada могут адресовать как динамическую, так и статическую память, обслуживая динамические и статические объекты данных. Тем самым расширяется область их применения [26, 52, 62].

Объявление указательного типа имеет следующий формат:

```
type <имя_указательного_типа> is access <имя_адресуемого_типа>;
```

Рассмотрим пример. Положим, что существуют следующие вспомогательные типы и тип «запись» по имени `Человек`:

```
type Т_Пол is ( ЖЕНЩИНА, МУЖЧИНА );
    type Т_Рост  is range 0 .. 250;  -- в см.
    type Т_Вес is range 0 .. 150;
    type Человек is
        record
            Имя : String ( 1 .. 10 ) := ( others => ' ' );
            Рост : Т_Рост := 0;
            Вес : Т_Вес := 0;
            Пол : Т_Пол;
        end record;
```

Объявим указательный тип

```
type У_Человек is access Человек;
```

и введем указатель

```
У_Петр : У_Человек;
```

Теперь мы можем объявить и проинициализировать динамический объект:

```
У_Петр := new Человек'( "Петр", 178, 75, МУЖЧИНА );
```

В последнем операторе выражение `new Человек'("Петр", 178, 75, МУЖЧИНА );` возвращает *l*-значение динамически выделенной (под человека) области памяти. Для инициализации содержания области применен агрегат. Это можно записать в другой, более громоздкой форме:

```
У_Петр := new Человек;
У_Петр.all := Человек'( "Петр", 178, 75, МУЖЧИНА );
```

На применение указателей к статическим объектам наложены определенные ограничения. Сделано это из соображений безопасности.

Во-первых, статическая переменная должна «добровольно согласиться» на введение дополнительного, косвенного имени, которым станет ее адрес, хранимый в указателе.

Во-вторых, вводятся два режима косвенного доступа к переменной: полный (для чтения и записи) и ограниченный (только для чтения). Режим задается синтаксической формой объявления указательного типа.

Для объекта-добровольца введена особая форма объявления:

```
Люди : aliased Integer;
```

Термин `aliased` в объявлении обозначает, что разрешается получение адреса целого объекта `Люди`. Иными словами, пометка `aliased` показывает, что у объекта может быть несколько имен: одно обычное, другое — адрес в указателе. Если слово `aliased` в объявлении отсутствует, то доступ к объекту через указатель запрещается.

С другой стороны, объявлением

```
type P_Integer is access all Integer;
У_Люди : P_Integer;
```

вводится указательный тип `P_Integer`. Он позволяет объявить указатель `У_Люди`, который способен хранить адрес целого объекта (типа `Integer`), имеющего пометку `aliased`. В этом объявлении ключевое слово `all` означает, что через указатель можно обращаться как для чтения, так и для записи.

Для присвоения указателю `У_Люди` адреса объекта `Люди` записывают:

```
У_Люди := Люди'Access;
```

Здесь для получения адреса объекта используют его атрибутную функцию 'Access. После выполнения этого оператора указатель У_Люди содержит адрес объекта Люди.

Указатель У_Люди можно разыменовать и тем самым получить *r*-значение адресуемого объекта. Операция разыменования обозначается как .all. Например:

```
X := У_Люди.all;
У_Люди.all := Z; -- здесь разыменование приводит к l-значению
```

ПРИМЕЧАНИЕ

При обращении к компонентам записи постфикс .all разрешается опускать, то есть можно записывать: < Указатель >.< ИмяПоляЗаписи >.

Доступ к объекту с помощью указателя может быть ограничен только чтением. Для этого в объявлении указательного типа вместо термина all надо использовать термин constant. В следующем фрагменте доступ к объекту Люди с помощью указателя ограничен только чтением:

```
declare
    type P_Integer is access constant Integer;
    M : Integer;
Люди : aliased Integer;
    У_Люди : P_Integer;
begin
    Люди := 47;
    У_Люди := Люди'Access;
    M := У_Люди.all;
end;
```

В языке Ada разрешены указатели на процедуры и функции. Благодаря этому процедура или функция может быть параметром другой процедуры или функции. Например, процедура Применить применяет функцию, передаваемую как параметр, ко всем элементам массива. Покажем реализацию этой процедуры:

```
type P_Fun is access function ( Item : in Float ) return Float;
type Vector is array ( Integer range <> ) of Float;
procedure Применить ( F : in P_Fun; To : in out Vector ) is
    begin
        for i in To'Range loop
            To ( i ) := F ( To ( i ));
        end loop;
    end Применить;
```

ПРИМЕЧАНИЕ

Атрибутная функция 'Range возвращает диапазон индексов для объекта-массива. При вызове функции F автоматически выполняется разыменование. Явно это записывается так: F.all(To(i)). Такую запись явного разыменования используют лишь тогда, когда у вызываемой процедуры или функции нет параметров.

Первый параметр процедуры Применить может быть любой функцией со следующей сигнатурой:

```
function ( Item : in Float ) return Float;
```

Такими функциями являются:

```
function Square ( f : in Float ) return Float is
    begin
        return f*f;
    end Square;
function Cube ( f : in Float ) return Float is
    begin
        return f*f*f;
    end Cube;
```

ПРИМЕЧАНИЕ

Мы вынуждены несколько «опередить события». Спецификация процедур и функций подробно объясняется в главе 13.

Массивы и указатели в языках C и C++

Операция разыменования указателя `p` в языке C обозначается так `*p`. Это одноместная операция. Применяя ее к указателю, получаем объект, на который указатель ссылается [14, 18, 30, 34].

Объявления же указателей в языке C имеют вид:

```
int *pa;
float *pb;
```

Форма такого объявления задумывалась как мнемоническая — специально для облегчения его понимания и использования. Первое объявление означает, что выражение `*pa` имеет тип `int`. Иными словами, результат разыменования указателя `pa` имеет тип `int`.

После обработки этих объявлений указатель `pa` может хранить адреса целых переменных, а указатель `pb` — адреса вещественных переменных.

Использование массивов и указателей в C тесно связано. Фактически имя массива `A` является указателем на его начальный, нулевой элемент `A[0]`. Таким образом, использование указателей для доступа к обычным массивам, размещенным в статической памяти, здесь считается обычным приемом.

Мало того, разрешена (и приветствуется) адресная арифметика, то есть манипулирование адресами, находящимися в указателях. Рабочим инструментом адресной арифметики считается адресное выражение.

Адресное выражение — это выражение, значением которого является адрес элемента массива (или какой-то ячейки памяти). Так, адресное выражение `A + 1` (значение указателя `A` плюс единица) указывает на элемент массива `A[1]`, а `A + `i указывает на `A[`i`]`. Все это приводит к очень интересным возможностям.

Приведем иллюстрирующий пример. Если имеются следующие объявления

```
int i;
int vector [100];
int *ptr;
```

то после выполнения оператора присваивания

```
ptr = vector;
```

имя указателя ptr становится эквивалентно имени массива vector и можно сделать следующие заключения:

- имя ptr[i] обозначает ту же ячейку массива, что и vector[i];
- разыменование адресного выражения *(ptr + i) дает значение, которое одновременно является *r*-значением индексированной переменной vector [i].

Иначе говоря, указатель на массив можно индексировать так, словно он является именем массива, а имя массива может служить компонентом адресного выражения.

При применении адресной арифметики следует соблюдать предельную осторожность. Дело в том, что удельный вес адресной единицы — это величина переменная. Она зависит от типа адресуемого элемента. Например, для типа char удельный вес равен единице, а для типа float — четырем.

Указатель в языке C может обслуживать любую статическую переменную с именем. Речь идет о том, что переменная помимо прямого имени может получить и косвенное имя — адрес, хранящийся в указателе.

В этих условиях требуется средство для определения адреса любой переменной. Роль такого средства играет операция взятия адреса, обозначаемая значком &. Применение этой операции к имени переменной возвращает адрес переменной, который можно сохранить в некотором указателе.

Например, если объявлена целая переменная с именем x и указатель на целые переменные p

```
int x;
int *p;
```

то после выполнения оператора

```
p = & x; /* запись адреса переменной x в указатель p */
```

операторы

```
*p = *p + 1;
x = x + 1;
```

считаются эквивалентными.

ПРИМЕЧАНИЕ

*p в правой части оператора присваивания обозначает значение x, а в левой части — место размещения (адрес) x.

Пример 1. Линейный поиск в массиве с использованием сигнальной метки x может быть описан следующим фрагментом.

```
a[0] = x; /* запись сигнальной метки */
i = n; /* стартовое значение индекса (ищем с конца) */
while ( a[i] != x )
    --i;
return i;
```

Запись этого же фрагмента на основе указателей имеет вид:

```
a[0] = x;
p = a + n;   /* p — указатель на последний элемент массива */
while (*p != x )
    --p;
return p - a;   /* p = a + i, поэтому, чтобы вернуть i, нужно вычесть a */
```

Теперь рассмотрим возможности языка C по представлению и обработке такой популярной структуры данных, как символьные строки.

Вспомним, что строки в языке C представляются как массивы из символов. Конец строки маркируется константой `EOS`, определяемой в виде `'\0'`.

Например, три символьные строки

```
маша
мыла
раму
```

хранятся в «ленте» памяти так, как это изображено на рис. 11.12.

Рис. 11.12. Лента памяти, хранящей три строки

Здесь подразумевается, что каждый символ занимает отдельную ячейку памяти.

Пример 2. Скопируем строку `buffer` в область памяти `start`, начиная с ячейки `start[next]`. Используем два указателя `p` и `q`.

```
p = &start [next] ;
q = buffer ;
for ( ; ; )
    {
        *p = *q ;
        if (*p  = =  EOS )  break ;
        p ++ ;
        q ++ ;
    }
```

Прокомментируем первый оператор программного фрагмента. Когда имя массива (являющееся указателем) связывается индексом, оно превращается в простое имя ячейки массива. Иначе говоря, от имени `start` взять адрес нельзя (это указатель), а от имени `start [next]` — можно.

После двух первых операторов `p` указывает на место, куда должен быть скопирован первый символ, а `q` указывает на первый элемент массива `buffer`.

Читаем `for ( ; ; )` как «всегда». Операторы между фигурными скобками повторяются до момента выполнения оператора `break`. Присваивание

```
*p = *q;
```

обеспечивает копирование одного символа.

Если это символ `EOS` (конец строки), тогда цикл завершается с помощью `break`. В противном случае с помощью постфиксной операции ++ увеличиваются `p` и `q`. Они указывают на следующий доступный элемент `start` и следующий копируемый символ соответственно.

Указатели языков C и C++ также могут адресовать функции, поддерживая возможность передавать в одни функции другие функции (в качестве параметров). Указатели крайне важны и для двустороннего связывания различных функций по данным.

Особую роль в C и C++ играют «бестиповые» указатели вида `void *`, обеспечивающие адресацию объектов любых типов. Они очень удобны при решении таких задач операционной системы, как управление памятью компьютера. Как правило, здесь идет речь о пересылке порций данных из одного места в другое,

невзирая на их форматы и внутреннюю структуру. Иными словами, перемещаться должны не числа или строки, а некие байты какой-то информации. Внутреннее содержание и структура «посылок» систему не интересует. Вот такими «неинтересующимися» почтальонами и служат бестиповые указатели. Эти «адресные механизмы» даже при желании не смогли бы выяснить содержание посылок, так как разыменовать бестиповой указатель просто нельзя. Конечно, после прибытия на место назначения данные могут быть расшифрованы, но к этому бестиповые указатели прямого отношения не имеют. Свое дело они уже сделали: сняли все запреты на пересылку.

Динамическое распределение памяти

В языке С для работы с динамическим объектами используют библиотечные функции `malloc( )` и `free( )`. Перед их применением к программе должен быть подключен заголовочный файл <stdlib.h>. Функция `malloc( )` выделяет память, а `free( )` — освобождает ее. Это значит, что при каждом запросе функция `malloc( )` выделяет требуемую область памяти в куче, а `free( )` освобождает ее, то есть возвращает область в пул памяти [14].

Заголовок функции `malloc( )` имеет вид:

```
void *malloc(size_t количество_байтов);
```

Здесь `количество_байтов` — это размер памяти, необходимой для размещения динамического объекта. Тип `size_t` задан в <stdlib.h> как некоторый целый без знака. Функция `malloc()` возвращает указатель вида `void *`, поэтому его можно присвоить указателю любого типа. В случае успеха `malloc( )` возвращает указатель на первый байт области памяти в куче. Если же в динамической распределяемой области памяти недостаточно, то память не выделяется и `malloc( )` возвращает нуль.

Для выделения 256 байтов следует записать:

```
char *ptr;
ptr = malloc(256); /* выделение 256 байтов */
```

В итоге присваивания указатель `ptr` содержит адрес выделенной области памяти.

Поскольку динамически распределяемая область памяти не бесконечна, следует проверять успешность размещения динамического объекта, например, следующим программным кодом:

```
p = malloc(128);
if(!p) {
    printf("Памяти не хватает!\n");
    exit(1);
}
```

Функция `free( )` выполняет противоположное действие: она возвращает в пул область памяти, выделенную ранее по функции `malloc( )`. Иначе говоря, она освобождает область памяти, которая может быть вновь использована функцией `malloc( )`. Функция `free( )` имеет следующий заголовок:

```
void free(void *p)
```

Здесь `p` — указатель на область памяти, выделенную перед этим функцией `malloc( )`.

ВНИМАНИЕ

Вызов функции free( ) с ошибочным аргументом мгновенно разрушит всю систему динамического распределения памяти.

В языке С++ для работы с динамическими объектами существуют операторы `new` и `delete[]`. Приведем пример:

```
int *ptr1;
int *ptr2 = new int[256]; // Размещение динамического массива
ptr1 = ptr2;
delete [] ptr2;           // Удаление динамического массива
```

Ничего «радостного» мы не показали. В результате этих манипуляций оба указателя стали повисшими, со всеми вытекающими отсюда последствиями.

Гибкость указателей в языке С

В отличие от языка Pascal аппарат указателей в С ориентирован не на обеспечение максимальной безопасности, а на предоставление программисту максимальной гибкости [14, 79]. В этом смысле язык С подобен острой бритве, которая хороша в умелых руках, но которой может сильно порезаться неумелый новичок.

1. В С указатели используют для доступа ко всей памяти, то есть для доступа как к обычным переменным (статическим, с именами), так и к безымянным переменным.
2. Выделение и освобождение динамической памяти не определяется средствами языка С (в отличие от языка Pascal), здесь используются библиотечные функции `malloc`, `free`.
3. В С отсутствуют ограничения на вид и тип переменных, на которые могут указывать указатели. Допускается проведение арифметических операций над указателями (адресами). В результате возможны многочисленные утечки памяти, появление повисших указателей.

Перечислим операции с указателями в языке С:

1. Присваивание `p = & a`
2. Разыменование `x = *p`
3. Получение адреса указателя `&p`.
4. Увеличение указателя на единицу `++p` (перемещение к адресу следующего элемента массива). Заметим, что имеется в виду адресная единица, удельный вес которой может меняться (в зависимости от типа адресуемого элемента). Например, если адресуются элементы, размещаемые в четырех ячейках памяти, то, как показано на рис. 11.13, вес адресной единицы равен четырем.

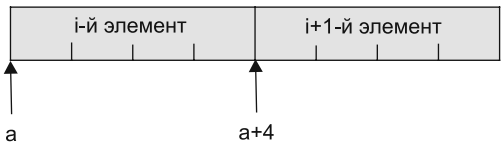


Рис. 11.13. Вес адресной единицы зависит от типа адресуемого объекта

5. Вычитание `p2 – p1`. Результат отображается в тех же единицах, что и размер данного типа переменной. Если `p2` и `p1` указывают на элементы типа `long`, то `p2 – p1 = 2` означает, что указываемые переменные разделены двумя значениям типа `long`, а не двумя байтами.

Ссылочный тип

Переменная ссылочного типа похожа на указатель, но имеет одну важную особенность. Указатель ссылается на адрес в памяти, а ссылка ссылается на объект или значение в памяти. Как следствие, совершенно естественно выполнять арифметические действия над адресами, но противоестественно делать то же самое со ссылками.

Язык C++ содержит специальную разновидность ссылочного типа, используемую преимущественно в формальных параметрах из объявления функции [18, 20, 34]. Переменная ссылочного типа в языке C++ — это константный указатель, который всегда неявно разыменован. Поскольку ссылочная переменная является константой, при определении она должна быть проинициализирована адресом некоторой переменной. После инициализации ссылочная переменная не может становиться ссылкой на какую-либо другую переменную. Неявное разыменование предостерегает от присваивания ссылочной переменной другого адреса. Иными словами, имя ссылочной переменной считается псевдонимом той переменной, которая использовалась при ее инициализации.

В определении имя переменной ссылочного типа предваряется символом амперсанда (&). Например,

```
int sum = 0;
int &ref_sum = sum;
. . .
ref_sum = 150;
```

В этом фрагменте переменные `sum` и `ref_sum` считаются алиасами.

Если ссылочные типы C++ используются в качестве типов формальных параметров для функций, то они обеспечивают два варианта взаимодействия между вызывающей и вызываемой функциями. В главе 13 подробно объясняется, что в языке C++ принят только один способ передачи параметров в функцию: передача по значению. Чтобы снять это ограничение, применяют указатели. Но указатели требуют явного разыменования, которое понижает читабельность и безопасность кода. В отличие от указателей ссылочные параметры записываются в вызываемой функции точно так же, как и остальные параметры. В вызывающей функции тоже не требуется особая форма определения того фактического параметра, которому соответствует ссылочный формальный параметр. В свою очередь, компилятор для ссылочного параметра передает адрес, а не значение.

В целях повышения безопасности создатели языка Java отказались от указателей в стиле C++. В отличие от ссылочных переменных C++, ссылочные переменные в Java могут применяться для ссылок на экземпляры различных классов, они не являются константами. Экземпляры всех классов в Java указываются ссылочными переменными. По сути, в языке Java используются только ссылочные переменные.

В следующем фрагменте `String` — это стандартный класс языка Java:

```
String str;
. . .
str = "Это строка в стиле языка Java";
```

В этом фрагменте `str` является ссылкой на экземпляр (объект) класса `String`. Изначально она установлена в `null`. Последующее присваивание превращает `str` в ссылку на объект класса `String`, содержащий строку `"Это строка в стиле языка Java"`.

Поскольку экземпляры классов в Java удаляются из кучи неявно (здесь нет явной операции удаления), в этом языке повисшие ссылки невозможны. В C# включены как ссылки в стиле Java, так и указатели в стиле C++. Однако применение указателей настоятельно не рекомендуется. Любая подпрограмма, использующая указатели, должна иметь модификатор `unsafe` (небезопасно). Объекты, указываемые ссылками, удаляются неявно, однако к объектам, доступным через указатели, механизм неявного удаления не применяется. Указатели включены в C# с единственной целью: обеспечить взаимодействие программ на C# с программами на C и C++.

В объектно-ориентированных языках Smalltalk, Python, Ruby и Lua все переменные являются ссылками. Все они неявно разыменованы. Более того, непосредственные значения этих переменных (адреса) просто недоступны.

Реализация указателей

Объект данных типа «указатель» реализуется областью памяти, в которой содержится адрес другой области памяти. Это начальный адрес области памяти, размещающей объект данных, на который ссылается указатель. Возможны два представления значений указателей в памяти:

- *Абсолютный адрес*. Значение указателя может быть истинным адресом области памяти, отведенной для объекта данных.
- *Относительный адрес*. Значение указателя может быть представлено как смещение от базового адреса некоторого более крупного блока памяти, называемого кучей, внутри которого размещается объект данных.

Если в указателе содержится абсолютный адрес, то объект данных, созданный операцией `new`, может быть размещен в любой области памяти компьютера. Обычно эта область располагается внутри общей кучи. Выбор нужного объекта с помощью абсолютного адреса весьма эффективен, поскольку обеспечивает непосредственный доступ к объекту данных с помощью аппаратных средств. Правда, при абсолютной адресации управление памятью усложняется, так как ни один объект данных не может быть перемещен в памяти, пока где-то существует на него указатель. Восстановление памяти, отведенной под объект данных, после того как он удаляется в ходе сбора мусора, также трудоемко, поскольку для каждого объекта данных освобождаемая память восстанавливается индивидуально, и блоки памяти должны быть возвращены обратно в общий пул свободного пространства памяти.

Если в указателях находятся относительные адреса, то предварительно назначаются блоки памяти, в пространстве которых и выделяются фрагменты памяти под создаваемые (с помощью операции `new`) объекты данных. Для каждого типа данных может формироваться своя отдельная область памяти. Впрочем, возможно

создание и единой области для объектов всех типов. В любом случае к каждой из областей применяются методы управления кучей. Если под объекты каждого типа отводится своя область памяти, то операция `new` располагает новые объекты в блоках памяти фиксированного размера, что упрощает процесс управления памятью. Выбор объектов данных при использовании относительной адресации подразумевает увеличение затрат по сравнению с абсолютной адресацией, поскольку для получения абсолютного адреса к каждому относительному адресу нужно добавлять базовый адрес области памяти, хранящей искомый объект. Преимущество относительной адресации заключается в возможности перемещения всей области в любой момент вычислений, без изменения значений указателя. Кроме того, вся выделяемая область памяти может рассматриваться как некий объект данных, который создается при входе в процедуру и используется для размещения объектов процедуры при помощи операции `new`, а затем удаляется при выходе из процедуры. При этом после удаления отдельного объекта не требуется реконструировать выделенную область, поскольку вся выделенная под такую область память восстанавливается при завершении работы процедуры.

Если каждый указатель ссылается на объекты только одного типа, то становится возможным статический контроль типов. Если же убрать это ограничение, то во время компиляции невозможно выяснить, на объект какого типа будет ссылаться указатель в ходе вычислений, поэтому придется осуществлять динамический контроль типов.

Специфика реализации указателей во многом определяется механизмом выделения памяти под новые объекты. Обычно применяют систему управления динамической памятью `heap`. В языке C, как правило, используется система управления памятью на основе стека, поэтому добавление функции динамического выделения памяти `malloc( )` требует существенного расширения всей структуры управления памятью в период вычислений.

Контрольные вопросы и упражнения

1. Объявите константу `size` равную 10.
2. Объявите массив `numbers` из 10 элементов типа `float`. Присвойте элементам следующие начальные значения `{0.0, 1.1, 2.2, 3.3, 4.4 …9.9}`.
3. Объявите указатель `ptrn`, который указывает на объект типа `float`.
4. Выведите на экран элементы массива `numbers`, используя запись индексов массива. Для организации цикла используйте структуру `for`, предварительно объявив целую переменную *i*. Каждое число нужно выводить с точностью до одного знака после десятичной точки.
5. Запишите два различных оператора, которые присваивают указателю `ptrn` начальный адрес массива `numbers`.
6. Выведите на экран элементы массива `numbers`, используя указатель `ptrn` плюс смещение.
7. Выведите на экран элементы массива `numbers`, используя имя массива плюс смещение.

8. Выведите на экран элементы массива `numbers`, индексируя указатель.
9. Сошлитесь на элемент 4 массива `numbers`, используя:
   - индекс массива;
   - имя массива (как указатель) плюс смещение;
   - указатель `ptrn` с индексом;
   - указатель `ptrn` плюс смещение.
10. Пусть начальный адрес массива равен `1002500`, длина каждого числа равна 4 байтам. Предполагая, что `ptrn` указывает на начало массива `numbers`, определить, на какой адрес ссылается выражение `ptrn+8`. Какое значение хранится по этому адресу?
11. Предполагая, что `ptrn` указывает на `numbers[5]`, определить, на какой адрес будет ссылаться `ptrn` после выполнения `ptrn -= 4`. Какое значение хранится по этому адресу?
12. Дано:

```
float  number1 = 7.3;
float  number2;
```

    а) объявить переменную `ptrf` как указатель на объект типа `float`;
    б) присвоить адрес переменной `number1` указателю `ptrf`;
    в) вывести на экран значение объекта, на который указывает `ptrf`;
    г) присвоить значение объекта, на который указывает `ptrf`, переменной `number2`;
    д) вывести на экран значение `number2`;
    е) вывести на экран адрес `number1`. Использовать формат `%lu` (длинное десятичное целое без знака `unsigned long int`). Совпадает ли его значение с адресом `number1`?
    ж) вывести на экран адрес, хранимый в `ptrf`. Использовать формат `%lu`. Совпадает ли его значение с адресом `number1`?

13. Пусть дано:

```
int *ptrz; /* будет указывать на массив z */
int *ptra = null;
void *ptrs = null;
int number, i;
int z[5] = {1, 2, 3, 4, 5,};
ptrs = z;
```

    Найти ошибки в следующих программных сегментах:

```
а)    ++ptrz;
б)    /*использование указателя для получения первого значения массива*/
      number = ptrz;
в)    /*присваивание переменной number значения элемента массива
        с индексом 2 (значения 3)*/
      number =*ptrz[2];
г)    /*вывод всего массива z*/
      for (i = 0; i <= 5; i++)
      printf("%d", ptrz[i]);
д)    /*присваивание значения, указываемого с помощью ptrs
        целой переменной number*/
      number = *ptrs;
ж)    ++z;
```

Глава 12
Преобразования типов данных

При разработке программ достаточно часто приходится решать вопрос о совместимости форматов значений различных объектов данных. В ходе подобных решений рассматривают три возможности: эквивалентность типов данных, явное приведение и неявное приведение форматов значений. Все эти возможности обсуждаются в текущей главе. Здесь поясняется структурная и именная форма эквивалентности, причем обсуждается как сильная (строгая), так и слабая (нестрогая) именная эквивалентность. Явные приведения форматов требуют участия программиста, а неявные приведения система выполняет самостоятельно. Далее разбираются достоинства и недостатки перечисленных разновидностей приведения. В качестве примера дополнительно описываются явные приведения в языке C++. В конце главы оцениваются уровни типизации популярных языков программирования, свидетельствующие об их защищенности от ошибок типов.

Эквивалентность типов данных

Тип данных может и должен рассматриваться как мощный инструмент контроля выполнения программы. Конечно, типы переменных позволяют компилятору определить размер необходимой памяти. Однако они же дают возможность проконтролировать правильность вычислений [12, 13, 17, 56, 80, 83, 88].

Вопрос об эквивалентности типа возникает при проверке выражений и операторов. Например, при вызове операции сравнивается тип данных переданного аргумента с предусмотренным для этой операции типом. Если эти типы одинаковы, то аргумент принимается и операция выполняется. Если эти типы различны, то либо создавшаяся ситуация расценивается как ошибка, либо применяется приведение типа аргумента к предусмотренному типу.

Правила эквивалентности типов просты и точны для предопределенных скалярных типов. Однако в случае составных типов, таких как массивы, записи и некоторые типы, определенные пользователем, правила существенно усложняются.

Возможны два вида эквивалентности:

- структурная эквивалентность;
- именная эквивалентность.

При структурной эквивалентности два типа считаются одинаковыми, если имеют одинаковую внутреннюю структуру, то есть состоят из *одинакового соединения*

одинаковых компонентов. *Структурная эквивалентность типов* означает, что две переменные имеют эквивалентные типы, если их типы обладают идентичной структурой.

Структурная эквивалентность используется в языках: Алгол 68, Модула 3, C, ML, ранний Pascal.

При именной эквивалентности каждое новое определение типа вводит новый тип (с новым именем). Если программист записал определения двух типов, то эти типы различны. В общем случае, *именная эквивалентность типов* означает, что две переменные имеют эквивалентные типы в любом из двух случаев:

- они определены в одном и том же объявлении;
- в их объявлениях используется одно и то же имя типа.

Именная эквивалентность наиболее популярна в современных языках. Она применяется в стандартном языке Pascal, языках Ada, C++, C# и Java.

Возможны некоторые вариации в трактовке этих двух подходов к эквивалентности, а многие языки используют комбинацию из данных подходов.

Точное определение структурной эквивалентности меняется от языка к языку. Оно зависит от решения: какие различия между типами считать важными, а какие — второстепенными. Например, многие считают, что формат объявления типа значения не имеет. Потому типы с объявлениями

```
type primer1 = record
    a, b : integer;
    end;
type primer2 = record
    a : integer;
    b : integer
    end;
```

структурно эквивалентны.

Но что сказать об определении:

```
type primer3 = record
    b : integer;
    a : integer
    end;
```

Должен ли реверс порядка полей изменять тип? Ответ неясен.

Наверное, различные представления констант здесь не следует принимать во внимание и следующие определения можно считать эквивалентными

```
type str1 = array [1..10] of char;
type str2 = array [1..2 * 5] of char;
```

С другой стороны, они, вероятно, отличны от

```
type str3 = array [0..9] of char;
```

Здесь длина массива не изменяется, но индексные величины различны.

Для определения структурной эквивалентности двух типов компилятор должен раскрывать их определения, рекурсивно замещая имена включенных типов соответствующими определениями до тех пор, пока не образуется строка из конструкторов типа, имен полей и встроенных типов. Если раскрытые строки одинаковы, то и типы структурно эквивалентны.

Структурная эквивалентность — это простой, но низкоуровневый подход к обсуждению типов, ориентированных на реализацию. Она не позволяет различать типы, о которых программист думает как о различных, но внутренняя структура которых случайно одинакова:

```
type  student = record
        name, address : string;
        age : integer
    end;
type  school = record
        name, address : string;
        age : integer
    end;
x : student;
y : school;
x : = y; -- ?
```

Большинство программистов захотят узнать об ошибке назначения типа `school` переменной типа `student`, но компилятор, чья проверка типов основана на структурной эквивалентности, допустит подобное присваивание.

Недостатки структурной эквивалентности:

- трудно определить эквивалентность типов со сложной структурой;
- частая проверка эквивалентности типов со сложными структурами может значительно увеличить стоимость компиляции;
- типы с различными ролями и одинаковой внутренней структурой неразличимы. Когда несколько программистов пишут общую программу, непреднамеренное совпадение типов различных объектов может привести к утрате всех преимуществ статической проверки типов, поскольку многие ошибки типизации остаются незамеченными.
- Нет общепринятых правил для принятия следующих решений: должны ли совпадать имена полей записи, или достаточно только совпадения их типов, количества и порядка следования? Если же имена полей записей должны совпадать, требуется ли тогда совпадение их порядка? Должны ли совпадать диапазоны значений индексов для массивов, или достаточно только совпадения числа элементов? Должны ли совпадать литералы и порядок их следования для двух перечислений?

Именная эквивалентность основывается на допущении, что если программист записал определения двух типов, то вероятно, что эти определения представляют различные типы! При именной эквивалентности полагают, что типы `X` и `Y` должны быть различны.

Обсудим следующий пример:

```
program main (input, output);
type M1: array [1..10] of real;
    M2: array [1..10] of real;
var a, c: M1; b: M2;
procedure Sub (x: M1);
    …
end;
begin              -- главная программа
        a := b;
```

```
        Sub (b)
end.
```

Поскольку два типа данных эквивалентны, если они имеют одинаковое имя, `M1` и `M2` считаются различными типами, даже если объекты данных этих типов имеют одинаковую структуру. В силу этого вызов `Sub (b)` некорректен. Кроме того, правильным будет оператор присваивания `a := c`, но не оператор `a := b`.

При структурной эквивалентности типы `M1` и `M2` будут считаться эквивалентными.

Эквивалентность имен как средство определения эквивалентности типов имеет следующие недостатки:

1. Тип каждого объекта, используемого в операции присваивания, должен иметь определенное имя. *Анонимные типы* не допускаются. Объявление языка Pascal

   ```
   var M3: array [1…10] of real;
   ```

   однозначно определяет тип массива `M3`, но переменная `M3` не может быть передана процедуре в качестве фактического параметра, поскольку ее тип не имеет имени.
2. Единственное определение типа должно использоваться во всей программе, поскольку тип объекта данных, переданного в качестве фактического параметра по цепочке процедур, не может заново определяться в каждой процедуре; должно обеспечиваться единое глобальное определение типа.

Практическое применение именной эквивалентности выдвигает вопрос: что делать с типами-алиасами? Должны ли считаться различными типы, когда определение одного из них состоит из имени другого? Имеется в виду следующее определение:

```
type A = B;
```

где `B` — имя другого типа.

Выделяют две разновидности именной эквивалентности:

строгая именная эквивалентность (алиасные типы рассматриваются как различные);

нестрогая именная эквивалентность (алиасные типы рассматриваются как эквивалентные).

Большинство языков, производных от языка Pascal, используют нестрогую именную эквивалентность. Лучшее решение предлагает язык Ada, в нем различают две разновидности алиасных типов [26, 52, 62]:

- подтипы (они совместимы с родительскими — базовыми типами);
- производные типы (они несовместимы с родительскими — базовыми типами).

Объявление подтипа начинается с ключевого слова `subtype`, а в объявлении производного типа задействован конструктор `is new`. Подтип и производный тип могут ограничивать свои значения относительно базового типа, для этого в их объявление вводится ключевое слово `range`, за которым указывается допустимый диапазон значений.

Приведем примеры объявлений на языке Ada:

```
subtype stack_element is integer; -- подтип совместим с базовым типом integer.
type celsius_temp is new integer; -- производный тип, не совместим
-- с базовым типом integer.
```

продолжение ⇩

```
type fahrenheit_temp is new integer;-- производный тип, не совместим
-- с базовым типом integer.
```

Обсудим следующие объявления:

```
type derived_type is new integer range 1..100;
subtype small_type is integer range 1..100;
```

Оба типа (`derived_type` и `small_type`) имеют одинаковый диапазон значений и наследуют все операции базового типа `integer`. Однако тип `derived_type` не эквивалентен типу `integer`. Напротив, тип `small_type` считается эквивалентным не только типу `integer` (при учете ограничения значений), но и любым другим подтипам от `integer`.

На экземпляры типа «неограниченный массив» в языке Ada распространяются правила структурной эквивалентности. Например, рассмотрим следующее объявление типа и объявления двух объектов:

```
type list is array (integer range <>) of integer;
list_1: list (1..10);
list_2: list (21..30);
```

Типы этих двух объектов эквивалентны, хотя диапазоны индексов у них представлены различными цифрами, поскольку к объектам типа «неограниченный массив» применяются правила структурной эквивалентности. Решающим фактором оказалось равенство *диапазонов* индексов (оба объекта включают по 10 элементов).

Для ограниченных анонимных типов Ada использует строгую именную эквивалентность. Рассмотрим следующие объявления на основе ограниченных анонимных типов:

```
m1 : array (1..10) of integer;
```

В этом случае `m1` получает анонимный, но уникальный тип, который назначается компилятором, но недоступен в программе. Если мы добавим

```
m2 : array (1..10) of integer;
```

то объекты `m1` и `m2` будут принадлежать к различным анонимным и не эквивалентным типам, хотя структурно эти типы идентичны. Множественное объявление

```
m3, m4 : array (1..10) of integer;
```

создает два анонимных типа, один для `m3`, а другой для `m4`, которые не эквивалентны. По сути, это объявление равносильно двум следующим объявлениям:

```
m3 : array (1..10) of integer;
m4: array (1..10) of integer;
```

Заметим, что в языке Ada правила именной эквивалентности строже того определения именной эквивалентности, которое мы ввели в начале раздела. Если же мы запишем

```
type vector is array (1..10) of integer;
m3, m4 : vector;
```

тогда типы для `m3` и `m4` станут эквивалентными.

Именная эквивалентность типов достаточно хорошо работает в языке Ада отчасти потому, что все типы, за исключением анонимных массивов, должны иметь явные имена (анонимные типы тоже получают внутренние служебные имена, они назначаются компилятором).

Правила эквивалентности типов в языке Ada достаточно четкие, чего не скажешь о тех языках, где разрешены многочисленные неявные приведения типов. Например, типы операндов сложения в языке Java могут быть любой комбинацией числовых типов. Просто тип одного операнда неявно приводится к типу другого. В языке Ada подобные неявные преобразования для операндов арифметических операций запрещены.

Язык C использует как именную, так и структурную эквивалентность типов. Объявление каждой структуры, перечисления и объединения создает новый тип с именем, который не эквивалентен любому другому типу. В силу этого, к типам «структура», «перечисление» и «объединение» применяется именная эквивалентность. Для других составных типов используется структурная эквивалентность. Типы массивов считаются эквивалентными, если состоят из компонентов одинакового типа. Так, тип массива с фиксированным размером эквивалентен типам массивов с таким же размером или типам без фиксированного размера. Полезно отметить, что оператор `typedef` в C и C++ не создает новый тип, он лишь вводит новое имя для существующего типа. Следовательно, любой тип, определенный с помощью `typedef`, эквивалентен родительскому типу. В правилах именной эквивалентности для C есть одно исключение: к структурам, перечислениям и объединениям, определенным в разных файлах, применяется структурная эквивалентность [79].

Язык C++ имеет похожие правила эквивалентности, здесь исключения для структур и объединений (из разных файлов) не предусмотрены [89].

В языках, где программистам не позволяют определять и именовать типы (Fortran, Cobol), именная эквивалентность неприменима в принципе.

Объектно-ориентированные языки (например, Java и C++) привнесли новую разновидность совместимости типов [44]. Она касается совместимости объектов и относится к иерархии наследования.

В заключение обсудим комплексный пример:

```
type cell = -- любое определение
type alink = указатель на cell
type blink = alink  -- это алиас
p, q : указатель на cell -- анонимный тип
r : alink
s : blink
t : указатель на cell
u : alink
```

При структурной эквивалентности все шесть переменных имеют одинаковый тип (указатель на `cell`).

При строгой именной эквивалентности `alink` и `blink` — разные типы, при нестрогой — они эквивалентны.

При строгой именной эквивалентности:

- `p` и `q` имеют разные типы, хотя и используют одно и то же анонимное (без имени) определение;
- `r` и `u` имеют одинаковый тип, поскольку используют одно определение (строка 2).

При нестрогой именной эквивалентности:

- `r`, `s`, `u` имеют одинаковый тип;
- `p` и `q` имеют одинаковый тип.

Преобразование типа и явное приведение

Существует много случаев, в которых ожидаются величины определенных типов. В операторе

```
a := выражение;
```

мы ожидаем в правой части тот же тип, который имеет `a`. В выражении `a + b` перегружаемый символ + означает или целое сложение, или вещественное сложение, поэтому мы ожидаем, что числа `a` и `b` являются либо целыми, либо вещественными.

При вызове процедуры `выч (арг1, арг2, арг3)` мы ожидаем, что типы аргументов будут совпадать с типами формальных параметров, объявленных в заголовке процедуры.

Предположим, что в каждом из этих случаев ожидаемые и обеспечиваемые типы должны быть одинаковы. Тогда, если программист захочет использовать значение одного типа в контексте, где ожидается другой тип, ему придется определить явное преобразование типа (иногда называемое *явным приведением*, *casts*).

ВНИМАНИЕ

Использование значения одного типа в месте, где ожидается другой тип, требует выполнения **явного преобразования типа** (явного приведения). Его **явно** задает программист.

Для преобразования может потребоваться программный код, выполняемый в период работы программы. Этот код будет зависеть от используемых типов.

Возможны три варианта:

1. Рассматриваемые типы структурно эквивалентны, но язык использует именную эквивалентность. В этом случае типы используют одинаковое низкоуровневое представление и имеют одинаковый набор значений. Поэтому приведение будет чисто концептуальной операцией и дополнительный код не нужен.
2. Типы имеют разные наборы значений, но пересекающиеся значения представляются одинаково. Например, один тип является поддиапазоном другого, или один содержит целые со знаком, а другой — целые без знака. Если обеспечиваемый тип имеет некоторые значения, которых нет у ожидаемого типа, тогда в период вычислений должен выполняться программный код, который гарантирует, что текущее значение корректно и для ожидаемого типа. Если проверка не проходит, то фиксируется динамическая семантическая ошибка. Если проверка успешна, то текущее обеспечиваемое значение применяется без изменений. Некоторые реализации языка могут разрешать отключение проверки. В итоге повышается скорость работы, но используется потенциально опасный код.
3. Типы имеют разные машинные представления, но можно определить некоторое соответствие между их значениями. Например, 32-разрядное целое может быть конвертировано в число с плавающей точкой двойной точности без потери точности. Большинство процессоров обеспечивают машинную команду для такого преобразования. Число с плавающей точкой может быть конвертировано в целое путем округления или усечения, но в дробной части будут потеряны некоторые

цифры, а для многих степенных значений произойдет переполнение. Заметим, что современные процессоры имеют машинную команду для такого преобразования. Преобразование между целыми числами различной длины достигается отбрасыванием разрядов или расширением знаком (старших разрядов).

Возможные преобразования между значениями целых и вещественных числовых типов иллюстрирует рис. 12.1.

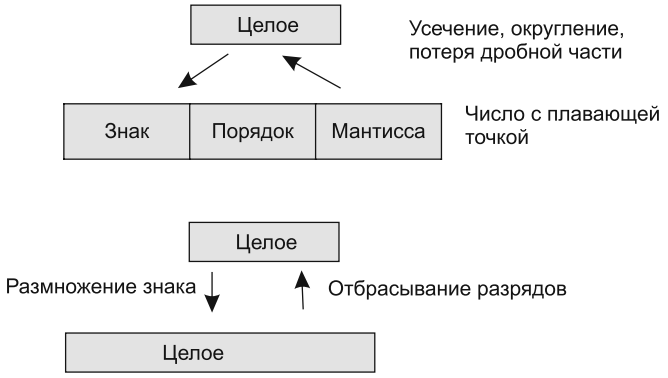


Рис. 12.1. Преобразования между значениями целых и вещественных типов

Напомним, что числа с плавающей точкой в формате IEEE имеют следующую структуру:

- Обычная точность — 1 бит (знак), 8 битов (порядок), 23 бита (мантисса).
- Двойная точность — 1 бит (знак), 11 битов (порядок), 52 бита (мантисса).

Рассмотрим примеры явных приведений типа в языке Ada.

```
type celsius_temp is new integer; -- объявление производного типа
subtype test is integer range 1..32; -- объявление подтипа, ограниченного
                                     -- 32 значениями
n : integer; -- длина 32 бита
r : real;    -- формат IEEE, двойная точность
t : test;
c : celsius_temp; --

   …
1.    t := test (n);    -- требуется семантическая проверка периода выполнения
2.    n := integer (t); -- проверка периода выполнения не нужна
3.    r := real (n);    -- требуется преобразование периода выполнения
4.    n := integer (r); -- нужны преобразование и проверка периода выполнения
5.    n := integer (c); -- не нужен код периода выполнения
6.    c :=  celsius_temp(n); -- не нужен код периода выполнения
```

В каждой из последних шести строк имя типа, в формат которого надо преобразовать значение, используется как *псевдофункция*, которая якобы выполняет преобразование типа. Первое преобразование требует проверки во время вычислений (чтобы гарантировать, что значение `n` находится в границах, разрешенных для подтипа `test`). Второе преобразование не требует программной проверки, так как каждое возможное значение `t` приемлемо для `n`. Третье и четвертое преобразования рассчитаны на применение кода, обеспечивающего изменение низкоуровневого

представления величин. При четвертом преобразовании дополнительно нужна семантическая проверка. Дело в том, что преобразование значения с плавающей точкой в целое значение приведет к потере цифр дробной части, эта потеря не считается ошибкой. Если же преобразование приведет к переполнению целого, то фиксируется ошибка. Два последних преобразования не нуждаются в программном коде периода выполнения. Типы `integer` и `celsius_temp` имеют подобные наборы значений и базовое представление [26].

Иногда, скажем, в служебных программах операционной системы, требуется изменить тип значения без изменения базовой реализации. Иначе говоря, нужно интерпретировать биты значения одного типа так, будто они являются значениями другого типа. Это требуется при программировании алгоритмов распределения памяти, которые обрабатывают большой массив символов или целых для представления кучи, а затем интерпретируют порции данного массива как указатели и целые (для регистрации использования системных ресурсов) или как отдельные структуры данных. В высокопроизводительных системах возникает нужда в представлении числа с плавающей точкой в виде трех целых полей (для знака, порядка и мантиссы) и параллельной обработке этих полей.

Изменение типа, которое не затрагивает базовые биты значения, называют *непреобразующим явным приведением типа*. Оно используется, например, в языке C++. Достаточно упомянуть оператор `reinterpret_cast`, реализующий непреобразующее явное приведение типа [34].

Явное приведение типа в языке C обозначается в виде взятого в скобки имени целевого типа, то есть того типа, к формату которого нужно привести значение:

```
int n;
float r;
r := (float) n; /* генерирует код для преобразования периода выполнения */
n := (int) r;   /* еще одно преобразование периода выполнения;
                   без проверки переполнения */
```

Непреобразующее приведение типа в C достигается взятием адреса от объекта, преобразованием типа полученного указателя и последующим разыменованием:

```
r =  * ((float *) & n);
```

Этот трюк работает, так как указатель на целые значения и указатель на значения с плавающей точкой имеют одинаковое представление (оба представляют адрес).

Явные приведения типа в языке C++

Современная версия языка C++ предлагает четыре специальных оператора для выполнения явного приведения типа: `static_cast`, `const_cast`, `reinterpret_cast` и `dynamic_cast` [18, 20].

Синтаксис этих операторов достаточно необычен и имеет следующий вид.

```
имя_целевой_переменной = имя_оператора <целевой_тип> (имя_операнда);
```

Целевая переменная принимает преобразованное значение. В качестве имени оператора используется одно из названий: `static_cast`, `const_cast`, `reinterpret_cast` или `dynamic_cast`. Имя в угловых скобках определяет целевой тип, к формату

которого требуется привести значение, а в круглых скобках записывается преобразуемый операнд.

Имя оператора определяет специфику приведения типа, применяемого к операнду.

ПРИМЕЧАНИЕ

Явные приведения типа можно задавать, используя эти операторы и как операции, то есть без левой части.

Оператор static_cast

Этот оператор обеспечивает обычные, статические приведения типа.

К примеру, для преобразования значения переменной `var_float` (типа `float`) в целое значение (типа `int`) мы можем записать:

```
var_int = static_cast<int>( var_float );
```

Приведение такого типа обычно осуществляется при присвоении значения большего арифметического формата переменной меньшего формата. В этом случае не исключена потеря точности результата. Обычно по этому поводу компиляторы выдают предупреждение. Когда приведение задано явно, предупреждений не будет.

Оператор `static_cast` используется также при выполнении преобразований, которые компилятор не способен выполнить автоматически. Например, его можно применить для возвращения значения указателя, которое было сохранено в указателе типа `void`:

```
double var_d = 247.0;
void *ptr = &var_d; //адрес любого объекта можно сохранить в указателе типа void
double *dp = static_cast<double *>(ptr); //преобразуем void обратно,
                                         //в исходный тип указателя
```

После сохранения адреса в указателе типа `void` впоследствии можно применить оператор `static_cast` и привести указатель к его исходному типу, что позволит восстановить доступ к динамическому объекту (ведь бестиповой указатель разыменовывать нельзя).

Оператор const_cast

Оператор `const_cast`, как и следует из его имени, преобразует тип операнда с пометкой `const` в аналогичный тип, но без пометки `const`. Заметим, что исходный тип изменению не подвергается. Все остальные атрибуты «типа-источника» и «типа-приемника» приведения должны совпадать.

Назначение модификатора `const` при объявлении указателя может сбивать с толку, потому что в зависимости от его расположения может означать одно из двух. В приведенных ниже строках описаны оба варианта:

```
const int *ptr_int;   // указатель на константу
int *const ptr_const; // константный указатель
```

Используя первый вариант объявления указателя, нельзя изменять значение переменной, на которую указывает указатель `ptr_int`, но можно изменять значение самого указателя `ptr_int`. Если применяется второй вариант, то все будет наоборот. Нельзя изменять значение самого указателя `ptr_const`, но можно изменять значение того, на что `ptr_const` указывает. Следует помнить различия в назначениях этих указателей, которые приведены в комментариях. Можно использовать `const` в обеих ролях и сделать константами как сам указатель, так и то, на что он указывает.

В случае константного приведения имеется в виду первый вариант (указатель указывает на константу).

Положим, что есть переменная типа `Key`, которая хранит пароль системы:

```
Key key;
```

В целях безопасности пользователю выдается указатель на ключ как на константу:

```
const Key *ptr_key;
```

Но пользователь хочет получить возможность изменять пароль, поэтому он создает свой указатель при помощи оператора `const_cast`:

```
Key *myPtr = const_cast<Key *>(ptr_key);
```

Теперь он может делать с паролем все, что угодно. Применение любой другой формы приведения в данном случае привело бы к ошибке при компиляции.

Оператор reinterpret_cast

Оператор `reinterpret_east` осуществляет низкоуровневую переинтерпретацию битов его операнда.

Оператор `reinterpret_cast` считается жестко машинно-зависимым. Для безопасного применения оператора `reinterpret_cast` следует хорошо понимать, как именно реализованы используемые типы, а также то, как компилятор осуществляет приведение. Обсудим следующий пример приведения:

```
int *ptr_int;
char *ptr_char = reinterpret_cast<char *>(ptr_int);
```

В подобном случае программист должен помнить, что фактическим типом объекта, адрес которого содержит указатель `ptr_char`, является `int`, а не символьный тип.

Любая попытка применения указателя там, где необходим обычный символьный указатель, скорее всего, потерпит неудачу именно в период выполнения. Например, его использование для инициализации объекта типа `string`, как в следующем случае, приведет к весьма необычному поведению во время вычислений.

```
string str(ptr_char);
```

Использование указателя `ptr_char` для инициализации объекта типа `string` — это наглядная демонстрации опасности явных приведений. Проблема заключается в том, что при изменении типа компилятор не формирует никаких предупреждений. При инициализации указателя `ptr_char` адресом для типа `int` компилятор не выдаст какого-либо сообщения об ошибке, поскольку задано явное преобразование. Однако любое последующее применение указателя `ptr_char` подразумевает, что он содержит адрес значения типа `char`. Компилятор не способен выяснить, что на

самом деле указатель хранит адреса целых чисел. Таким образом, инициализация строки `str` при помощи указателя вполне допустима, но по сути даже преступна. Отследить причину такой проблемы чрезвычайно трудно.

Оператор dynamic_cast

Оператор `dynamic_cast` применяется для преобразования ссылки или указателя на объект базового класса в ссылку или указатель на объект другого, родственного класса.

В отличие от других способов приведения, оператор `dynamic_cast` обеспечивает контроль соответствия типов в период вычислений. Если объект, связанный со ссылкой или указателем, не является объектом результирующего класса, оператор `dynamic_cast` приводит к ошибке. Если неудачу потерпит динамическое приведение указателя, операция `dynamic_cast` вернет значение 0. Если неудачу потерпит динамическое приведение ссылки, формируется исключение «`bad-cast`».

Класс объекта, к которому будет приведена ссылка или указатель, в период компиляции обычно не известен. В результате динамического приведения указатель или ссылка на объект базового класса заменяется указанием на объект производного класса.

Совместимость типов и неявное приведение

В конкретном контексте многие языки не требуют эквивалентности типов. Вместо этого говорят, что значения типов должны быть совместимы с тем контекстом, в котором они появляются.

Существует много случаев, в которых ожидаются величины определенных типов. В операторе присваивания тип в правой части должен быть совместим с типом из левой части. Типы операндов сложения должны быть совместимы или с целым, или с вещественным типом.

В операторе вызова подпрограммы типы аргументов должны быть совместимы с типами соответствующих формальных параметров.

От языка к языку определение совместимости сильно меняется. Язык Ada использует достаточно разумный подход [26, 52]. Здесь тип `S` совместим с типом `T`, если и только если:

1) `S` и `T` эквивалентны;
2) один из них является подтипом другого (или оба являются подтипами одного и того же базового типа);
3) оба являются массивами с одинаковым количеством и типами элементов по каждому измерению.

Pascal более мягок — дополнительно разрешается смешивание базовых типов и их поддиапазонов, что позволяет использовать целый тип в контексте, где ожидается вещественный тип.

Если язык позволяет использовать значение одного типа в контексте, где ожидается другой тип, то реализация языка должна выполнять автоматическое, неявное

преобразование к ожидаемому типу. Такое преобразование называется *неявным приведением типа* (*coercion*).

Как и при явных приведениях, при неявном приведении может потребоваться код периода выполнения:

1) для динамической семантической проверки;

2) для преобразования машинных представлений.

В языке Ada преобразования машинных представлений при неявных приведениях запрещены (из-за соображений безопасности). Рассмотрим пример для языка Ada.

```
d : weekday;
k : workday;
type calendar_column is new weekday;
c : calendar_column;
. . .
1. k := d; -- требуется проверка периода выполнения;
2. d := k; -- не требуется проверка; каждый рабочий день является днем недели;
3. c := d; -- статическая семантическая ошибка;
```

Первое преобразование называется *сужающим приведением*, поскольку оно уменьшает диапазон возможных значений (подразумевается, что тип `workday` имеет пять значений, а тип `weekday` — семь значений). В этом случае просто необходима проверка допустимости преобразования.

ВНИМАНИЕ

Такое преобразование особо опасно и противоречит основной концепции неявного приведения типов: не допускать потерю информации!

Напротив, второе преобразование называют *расширяющим приведением*, так как оно расширяет диапазон возможных значений.

Третье же преобразование просто запрещено. Типы `weekday` и `calendar_column` не совместимы, поскольку `calendar_column` является производным типом от `weekday`.

Для выполнения в языке Ada последнего присваивания нужно применить явное приведение:

```
c := calendar_column (d);
```

Сужающее приведение конвертирует значение в формат типа, у которого нет никаких аналогов для всех значений исходного типа. К примеру, конвертация типа `double` в тип `float` языка Java означает потерю существенного количества значений. С другой стороны, расширяющее приведение конвертирует значение в формат типа, у которого существенно больше значений. Это справедливо для случая конвертации значения из типа `int` в тип `float` языка Java. Расширяющее преобразование безопасно практически всегда, тогда как о сужающем этого сказать нельзя. Например, если в Java-программе значение с плавающей точкой `1.7E54` конвертируется в целое, то результат может существенно отличаться от оригинальной величины [44].

Потенциальной проблемой расширяющего приведения считают потерю точности представления. Во многих реализациях языков, например, целые числа являются 32-разрядными, что обеспечивает представление не менее девяти десятичных цифр.

Формат числа с плавающей точкой также составляет 32 разряда, но гарантирует представление лишь семи десятичных цифр (из-за необходимости сохранения порядка). Следовательно, переход от целого формата к формату плавающей точки может привести к потере двух цифр точности.

Достаточно сложны, конечно, неявные приведения составных типов. Много вопросов возникает при обеспечении совместимости массивов и записей в ходе присваивания. Комплексный характер имеет проблема переопределения родительских методов в дочерних классах: какие параметры следует учитывать в процессе переопределения?

Очень часто в программах используются смешанные выражения, в которых операнды некоторой операции имеют разные типы. В этих ситуациях должны определяться соглашения для неявного приведения типов операндов.

Увы, но единого мнения по неявному приведению типов операндов нет. Для иллюстрации «камня преткновения» рассмотрим следующий программный код на языке Java:

```
int a;
float x, y, z;
. . .
z = x * a;
```

Допустим, что вторым операндом умножения планировалась переменная `y`, но из-за ошибки при наборе была введена буква `a`. Поскольку в языке Java допустимы смешанные выражения, то компилятор не воспримет это как ошибку — он просто вставит команды, приводящие целую переменную `a` к типу `float`. Однако если бы смешанные выражения в языке Java не разрешались, то была бы сформирована ошибка типизации.

Поскольку смешанные выражения снижают безопасность, язык Ada существенно ограничивает смешивание типов операндов, в частности запрещая смешивание целых чисел и чисел с плавающей точкой. При записи этого же фрагмента для Ada

```
a : integer;
x, y, z : float;
. . .
z := x * a;
```

компилятор нашел бы ошибку выражения, поскольку операнды типов `float` и `integer` нельзя смешивать в операции умножения.

Очевидно, что с точки зрения проектирования языка вопрос формулируется так: кто должен находить в выражениях ошибки типизации — компилятор или программист?

Языки ML и F# запрещают неявные приведения операндов в выражениях. Все необходимые преобразования должны быть явными. В результате здесь обеспечивается такой же высокий уровень надежности в выражениях, как и в Ada.

В большинстве других языков ограничения на смешивание в арифметических выражениях отсутствуют.

В C-подобных языках предусмотрено семейство целых типов с более коротким форматом, чем в типе `int`. В языке Java это типы `byte` и `short`. Операнды этих типов неявно приводятся к типу `int` перед применением практически любой операции [79, 89]. Обсудим следующий фрагмент на языке Java:

```
byte x, y, z;
. . .
z = x + y;
```

Значения `x` и `y` неявно приводятся к типу `int`, а затем выполняется целочисленное сложение. Далее сумма конвертируется в тип `byte` и заносится в переменную `z`. Учитывая большую емкость памяти современных компьютеров, особого смысла в использовании типов `byte` и `short` просто нет.

Конечно, неявные приведения — спорный вопрос в проектировании языка. Основной аргумент противников заключается в том, что они позволяют смешивать типы *без явного указания программиста*, аннулируя выгоды проверки типов и снижая надежность программ. Сторонники же этих преобразований говорят о существенной потере гибкости, к которой приводит отказ от неявных приведений. Впрочем, Fortran и C, имеющие слабые системы типизации, разрешают целые категории неявных приведений. Здесь позволено смешивать значения большинства численных типов и выполнять неявные приведения типов (прямые и обратные) по мере необходимости.

Приведем несколько примеров для языка C.

```
short int s; /* длина переменной — 16 разрядов */
unsigned long int L; /* длина переменной — 32 разряда */
char c; /* может быть со знаком или без знака — зависит от реализации */
float f; /* обычно одинарная точность по IEEE */
double d; /* обычно двойная точность по IEEE */
. . .
1.    s = L; /* порядок битов в L интерпретируется как число со знаком */
2.    L = s; /* s расширяется знаком до большей длины, затем его биты
интерпретируются как число без знака */
3.    s = c; /* c расширяется или знаком или нулем до длины s, затем результат
интерпретируется как число со знаком */
4.    f = L; /* L преобразуется в формат с плавающей точкой, так как f имеет меньше
значащих битов, некоторая точность будет потеряна */
5.    d = f; /* f конвертируется в больший формат; нет потери точности */
6.    f = d; /* d конвертируется в меньший формат, точность может быть потеряна.
Если значение d не может быть представлено с одинарной точностью, результат не
определен, но нет динамической семантической ошибки */
```

Язык C не позволяет смешивать записи (структуры), если они не являются эквивалентными структурно, причем с идентичными именами полей. Язык C не обеспечивает операции, в которых операндами являются массивы, зато здесь разрешено (во многих случаях) смешивать массивы и указатели.

Наиболее современные языки отражают тенденцию перехода к статической типизации, отказываются от неявного приведения типов. Некоторые разработчики языков, напротив, говорят, что неявные приведения — естественный механизм, упрощающий расширяемость программ, применение новых типов (наряду со старыми). В частности, язык C++ предлагает предельно богатый, расширяемый программистом набор правил неявного приведения. При определении нового типа программист может задать операции неявного приведения для конвертирования значений нового типа в значения для существующих типов (и наоборот). Эти правила перекликаются с правилами для перегрузки операций, они добавляют языку существенную гибкость, но относятся к наиболее трудным для понимания понятиям.

Подводя итог, отметим наличие двух противоположных мнений относительно масштаба применения неявного преобразования типов. В языках Pascal и Ada оно почти не используется; любое несоответствие типов, за малым исключением, воспринимается как ошибка. В языке C приведение типов считается правилом — когда компилятор обнаруживает несоответствие типов, он пытается найти подходящую операцию преобразования, которую можно было бы вставить в программу для нужного изменения типа значения. Сообщение об ошибке возникает лишь тогда, когда компилятор не может произвести требуемое преобразование типов.

Несоответствие типов является часто встречающейся ошибкой, инициирующей необходимость в преобразовании типов; оно характерно для тех языков, где определено большое количество типов данных. Конечно, автоматическое приведение типов освобождает программиста от рутинной работы, которая потребовалась бы для явного введения в программу операций преобразования типов. С другой стороны, приведение типов может скрыть наличие других, более серьезных ошибок, которые в противном случае могли бы быть замечены при компиляции.

Уровень типизации языка

В общем случае проверка типа гарантирует, что операции в программе применяются правильно.

Цель проверки типа — защита от ошибок вычислений.

В данном контексте мы имеем в виду ошибки типизации. Такие ошибки происходят, если какие-то операции применены некорректно, например, если целое число ошибочно рассматривается как нечто другое. Поясним суть ошибок типизации. Пусть где-то в программе описана функция:

```
function name ( a : S ) : T;
```

Положим также, что в программе есть объявления переменных:

```
var b : Q; r : T;
```

И наконец, в разделе операторов указано использование этой функции:

```
r := name (b);
```

Увы, тип формального параметра не совпадает с типом фактического параметра: переменная `b` имеет тип `Q`, а не `S`. Это ошибка типизации, которую нужно выявить и устранить.

Программу, выполняемую без ошибок типизации, называют *безопасной по типам* (*type safe program*).

Возможны два варианта проверки типов:

- статическая проверка;
- динамическая проверка.

Статическая проверка проводится при компиляции. Ее объектом является весь исходный текст программы. В ходе статической проверки применяются правила системы типизации. При этом не требуется выполнение программы, просто просматривается текст, и для каждой функции (операции) анализируются операнды.

Динамическая проверка осуществляется во время выполнения программы (в период *runtime*). Здесь работает дополнительный программный код, вставляемый в программу для обнаружения ошибок. Кроме того, дескриптор каждого объекта данных несет в себе полную информацию об атрибутах типа, собранную на этапе компиляции.

При динамической проверке готовая к употреблению программа состоит из исходного текста и дополнительного кода, обеспечивающего проверку (рис. 12.2).

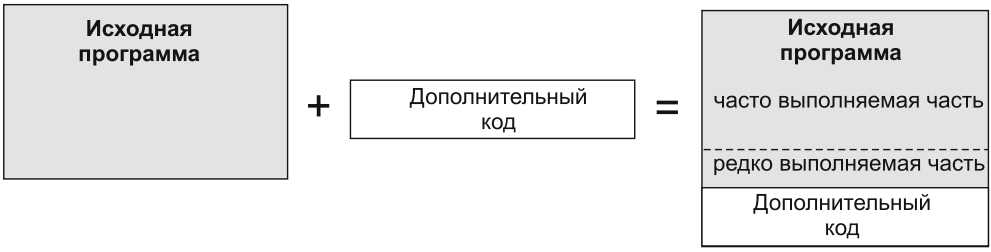


Рис. 12.2. Подготовка программы к динамической проверке

В этом случае отслеживается ход вычислений, регистрируются ошибки типов при выполнении конкретных операторов.

Серьезный недостаток динамической проверки состоит в том, что ошибки могут прятаться и проявляются только в момент работы операторов программы. Если ошибки находятся в редко выполняемой части программы, то и выявляться они будут редко. Кроме того, требуется дополнительное время для разработки и место для размещения дополнительного кода, необходима высокая квалификация программиста.

Статическая проверка достаточно эффективна, а динамическая проверка стоит столь дорого, что реализации классических языков обычно ориентируются лишь на статические проверки типов на основе исходного текста. Свойства программ, которые зависят от величин, вычисляемых в период выполнения, проверяются редко. Например, императивные языки редко проверяют, что индекс массива находится в пределах допустимых границ. Это характерно для традиционных языков программирования, обеспечивающих высокую надежность и скорость вычислений. В функциональных и скриптовых языках доминирует динамическая проверка, поскольку здесь главной целью считается высокая гибкость программирования.

По эффективности предупреждения ошибок выделяют сильные и слабые системы типизации.

Если при помощи статической проверки можно обнаружить все без исключения ошибки типов в программе, то система типизации такого языка называется *сильной*. Полномасштабная проверка типов является некоторой гарантией отсутствия соответствующих ошибок в программе. Функцию `f` с сигнатурой

```
f : S → R
```

называют *безопасной в отношении типа*, если при вычислении этой функции результат не может выйти за пределы множества `R`. В случае всех операций, безопасных в отношении типа, мы знаем еще до выполнения программы, что тип результата будет правильным и что динамическая проверка типов не требуется. Очевидно,

если каждая операция безопасна в отношении типа, то язык в целом является *сильно типизированным*.

Увы, лишь немногие языки соответствуют этому требованию. Например, в языке C, если `A` и `B` имеют тип `short` (являются короткими целыми числами), то результат операций `A + B` или `A * B` может оказаться за пределами этого типа, вследствие чего возникнет ошибка типа. Словом, реализация в языке настоящей, строгой проверки типов весьма трудна. Кстати, рассмотренные нами механизмы для ограничения преобразования типов существенно облегчают эту реализацию.

Сильная система типизации разрешает только безопасные выражения, которые гарантируют вычисления без ошибок типов. Система типизации считается *слабой*, если она не является сильной.

Сами по себе термины «сильный» и «слабый» дают мало информации.

Реально, в силу трудностей реализации, связанных с ресурсными и временными ограничениями при проектировании языка и создании компилятора (интерпретатора), сильная система типизации разрешает только некоторое подмножество безопасных программ (рис. 12.3).

Рис. 12.3. Диаграмма для языков с сильной типизацией

Однако у нас нет информации о размере этого подмножества. Если система типизации придерживается философии «лучше безопасно, чем извините», то она будет отвергать многие программы. Патологический пример — система типизации, которая отвергает все программы. Такая система будет сильной, но бесполезной.

Для слабой системы типизации характерна другая диаграмма (рис. 12.4).

Как видим, программистам на этих языках приходится довольствоваться узким сегментом пересечения областей безопасных и слабых (с точки зрения ошибок типизации) программ.

Язык Pascal — это почти сильно типизированный язык. Этим «почти» он обязан своему типу «вариантная запись», который позволяет опускать тег, определяющий текущий вариант и являющийся инструментом проверки корректности структуры переменной.

Язык Ada — сильно типизированный язык. Правда, нужно сделать некоторую оговорку. В нем имеется родовая библиотечная функция `Unchecked_Conversion` (Неконтролируемое преобразование), которая позволяет в любой момент отказаться от сильной типизации [52]. Заголовок этой функции имеет вид:

```
generic
type Source(<>) is limited private;
type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target;
```

где `Source` — определяет исходный тип, а `Target` — целевой тип данных.

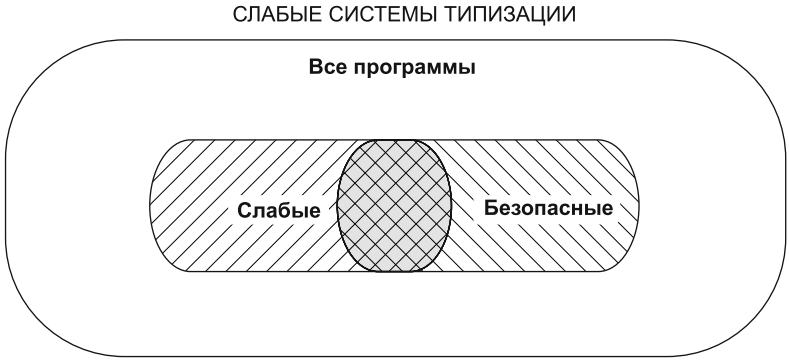


Рис. 12.4. Диаграмма для языков со слабой типизацией

Как видим, функция может быть настроена на любую пару типов данных. При настройке `Source` и `Target` заменяются именами конкретных типов. Настроенная функция принимает в качестве аргумента переменную типа `Source` и возвращает строку битов, представляющую собой текущее значение этой переменной в формате типа `Target`. Действительное преобразование при этом не выполняется: просто извлекается значение переменной одного типа, которое используется в качестве значения другого типа. Ее настроенная версия временно приостанавливает проверку конкретного типа. Подобная разновидность преобразования называется *неконвертирующим явным приведением*.

Неконтролируемое преобразование может быть полезно в операциях с памятью (размещение, освобождение), которые определяются пользователем. В этих операциях адреса рассматриваются как целые числа, а не как указатели. Поскольку в функции `Unchecked_Conversion` нет проверки типов, то за осмысленное использование переменной, полученной из этой функции, ответственность несет программист.

Языки C и C++ не считаются сильно типизированными языками, поскольку включают тип «объединение», типы параметров которых не проверяются.

Сильно типизированным языком является язык ML, хотя типы некоторых параметров для функций могут быть неизвестны в период компиляции. F# — сильно типизированный язык

Языки Java и C# сильно типизированы, в том же смысле, что и язык Ada. Типы могут приводиться явно, что может вызвать ошибку определения типа. Тем не менее все ошибки типизации выявляются.

Правила неявного приведения существенно влияют на качество проверки типов. Например, выражения в языке Java являются сильно типизированными. Тем не менее допускаются арифметические операции над парой операндов разного типа

(целый тип и тип с плавающей точкой). Выполняемое при этом неявное приведение снижает уровень обнаружения ошибок, а значит, и качество сильной типизации. Допустим, что в программе используются целые переменные `x` и `y`, а также вещественная переменная `z`. Если программист хотел набрать `x + y`, а ошибочно набрал `x + z`, то компилятор ошибку не обнаружит, а просто неявно приведет значение `x` к типу с плавающей точкой. Следовательно, неявное приведение отрицательно влияет на качество сильной типизации. Языки, в которых широко используется неявное приведение типов (C и C++), существенно уступают в надежности языкам с минимальным количеством неявных приведений (Ada) и языкам, где они вообще отсутствуют (ML и F#). Языки Java и C# разрешают вдвое меньшее количество неявных приведений, чем язык C++, поэтому диагностика ошибок в них лучше, но не столь эффективна, как в ML и F#.

Контрольные вопросы и упражнения

1. Для знакомого вам языка опишите, как в нем определена эквивалентность типов — как именная эквивалентность или как структурная эквивалентность? Рассмотрите каждый тип данных отдельно (поскольку способ определения эквивалентности для разных типов может быть разным) и точно объясните, когда две переменные соответствующего типа считаются имеющими одинаковый тип и когда переменная, представляющая фактический параметр, и формальный параметр процедуры считаются имеющими одинаковый тип. Различаются ли способы определения эквивалентности для простейших типов данных и для определяемых программистом?
2. Предложите три различных способа определения эквивалентности типа для *записей* и два способа для *массивов*, основываясь на понятии структурной эквивалентности.
3. Дайте определение явного приведения типов. Охарактеризуйте типовые ситуации, которые могут встретиться при необходимости явного преобразования типов.
4. Поясните смысл неявного приведения типов. Раскройте достоинства и недостатки этого механизма.
5. Выберите хорошо известный вам язык программирования. Для каждого типа данных этого языка определите использованный в нем механизм эквивалентности типов. Объясните, почему выбран именно этот механизм?
6. Для выбранного вами языка программирования определите вид системы типизации. Какие безопасные программы разрешает и отвергает система типизации данного языка?
7. В хорошо известном языке программирования выберите какой-либо составной тип данных. Определите, статическая или динамическая проверка типов применяется для проверки правильности использования всех операций, определенных для этого типа.

Глава 13
Подпрограммы

В данной главе внимание сосредоточено на детальном обсуждении обширного комплекса вопросов, связанных с подпрограммами (процедурами и функциями): от их определения до применения. Поясняются все основные способы передачи параметров в подпрограммы (и из подпрограмм), а также сопутствующие вопросы (организация области видимости, проверка типов параметров и т. д.). Объявления и применения подпрограмм иллюстрируются многочисленными примерами на различных языках программирования. Достаточно подробно освещена проблема полиморфизма подпрограмм во всех его проявлениях: параметрический полиморфизм, полиморфизм подтипов, перегрузка и неявное приведение. Проанализирована реализация полиморфизма в различных языках программирования. Помимо обычных подпрограмм, рассмотрены различные реализации родовых подпрограмм, обеспечивающих параметрический полиморфизм. Детально описан особый режим использования подпрограммы — замыкание.

Подпрограмма — это абстрактное действие, определяемое программистом. Подпрограммы являются теми базовыми блоками, из которых строится большинство программ, и почти в каждом языке можно найти средства для их объявления и вызова. Возможны два взгляда на подпрограммы. На уровне проектирования программы она интересна тем, что представляет абстрактное действие, определенное программистом, — в противоположность элементарным операциям и операторам, встроенным в язык. На уровне разработки языка нас интересует разработка и реализация общих средств объявления и вызова подпрограммы. Хотя эти два взгляда и пересекаются, удобнее рассматривать их отдельно.

Разновидности подпрограмм

Подпрограмма — программный сегмент многократного применения, задает определяемое программистом действие [12, 17, 86, 88].

Различают две формы подпрограмм:

- *функции* — они расширяют встроенные в язык операции;
- *процедуры* — они расширяют встроенные в язык операторы.

Для использования подпрограммы ее необходимо вызвать (рис. 13.1).

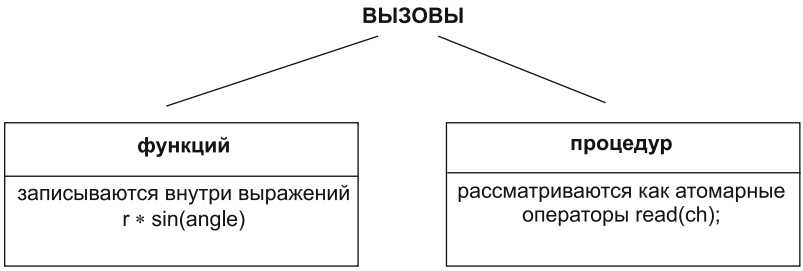


Рис. 13.1. Вызовы функций и процедур

Вызовы функций записываются внутри выражений. Например, вызов функции всегда помещается в правую часть оператора присваивания:

```
y := r * sin (angle) + … ; (* здесь вызывается функция для вычисления синуса *)
```

Вызов процедуры рассматривают как атомарный оператор:

```
read (x);
```

Вызов любой подпрограммы записывается в префиксной нотации:

```
<имя_подпрограммы> (<фактические параметры>);
```

Параметры в операторе вызова подпрограммы называют *фактическими*. Скобки вокруг параметров — это синтаксический признак вызова подпрограммы. Иногда необходимо указывать скобки в вызове подпрограммы, даже если параметров нет.

Оператор

```
begin   while EOLN do   readln;   read (ch)   end;
```

содержит три вызова:

1. `EOLN` — вызов функции без параметров (проверяет наличие признака конца строки при вводе);
2. `readln` — вызов процедуры без параметров (пропускает все оставшиеся элементы текущей строки ввода);
3. `read (ch)` — вызов процедуры с одним параметром `ch`.

Фактический параметр — это объект данных, совместно используемый вызывающей программой и вызываемой подпрограммой. Различают три разновидности фактического параметра:

- ❑ локальный объект данных, принадлежащий вызывающей программе;
- ❑ нелокальный объект данных, видимый из вызывающей программы;
- ❑ результат вычисления функции, вызванной из вызывающей программы, который возвращается в точку вызова.

В точке вызова подпрограммы фактический параметр может представляться *выражением фактического параметра*. Это обычное выражение, допустимое в языке программирования. Например, подпрограмма `Subprog` может быть вызвана с любой из разновидностей выражения фактического параметра, приведенной в табл. 13.1.

Таблица 13.1. Разновидности фактического параметра подпрограммы

Вызов подпрограммы в программе	Фактический параметр в программе
`Subprog(i, b)`	`i, b` — локальные переменные программы
`Subprog(a+b, x-y*z)`	`a+b, x-y*z` — арифметические выражения программы
`Subprog (45, true)`	`45, true` — константы
`Subprog (p1, p2)`	`p1, p2` — формальные параметры в программе. Здесь программа рассматривается как составная часть еще большей программы, которая считает ее своей подпрограммой
`Subprog (g1,g2)`	`g1, g2` — глобальные или нелокальные переменные в программе
`Subprog (m[i], x.a1)`	`m[i], x.a1` — элементы массивов и записей
`Subprog (sqrt(i), func(y))`	`sqrt(i), func(y)` — результаты вычисления элементарной функции и определенной программистом функции

Если при вызове подпрограммы ей передается параметр в форме выражения, то во время вызова, но перед входом в подпрограмму, вычисляется значение этого выражения. Объект данных, полученный в итоге вычисления выражений, становится фактическим параметром, передаваемым подпрограмме.

Фактические параметры несут в себе исходные данные, которые *фактически* обрабатываются подпрограммой. Этим и объясняется их название.

Еще раз повторим: функция играет роль *операции*, а процедура — роль *оператора*. Они похожи, хотя функция возвращает результат, а процедура — нет. Неудивительно, что в языке C обе разновидности подпрограмм вызываются с помощью функций, а в языке Модула 2 — с помощью процедур.

Объявление подпрограммы

Объявление подпрограммы описывает ее содержание и включает следующие элементы:

- имя;
- формальные параметры (обозначают места для подстановки фактических параметров);
- тип результата (отсутствует в случае процедуры);
- тело (объявления локальных переменных и операторы).

Обсудим функцию вычисления квадрата: `square(2)` ⇒ `4; square(4)` ⇒ `16`.

При использовании имени `x` для обозначения места подстановки фактического параметра вычисление в функции можно записать в виде:

```
square(x) = x * x;
```

Формальный параметр `x` обычно называется *меткой-заполнителем* (*placeholder*). Таким названием подчеркивают обязанность формального параметра: чаще всего он принимает значение фактического параметра, существующего в точке вызова.

Формальный параметр — это разновидность локального объекта данных в подпрограмме. Имя формального параметра представляет собой простой идентификатор, а в его объявлении обычно задают тип и другие атрибуты, как и в случае объявлений обычных локальных переменных. Например, в заголовке процедуры на языке C

```
void Subprog(int A; char B)
```

задается два формальных параметра `A` и `B` и объявляется, что один из них принадлежит к целому типу, а другой — к символьному. В общем случае объявление формального параметра несколько отличается от объявления переменной.

В зависимости от выбранного способа передачи формальный параметр может становиться псевдонимом фактического параметра, перенимать его имя (собственное имя формального параметра заменяется именем фактического параметра). В классическом варианте формальный параметр считается хранителем значения фактического параметра.

Пример объявления функции `square` (в синтаксисе языка Pascal) приведен на рис. 13.2.

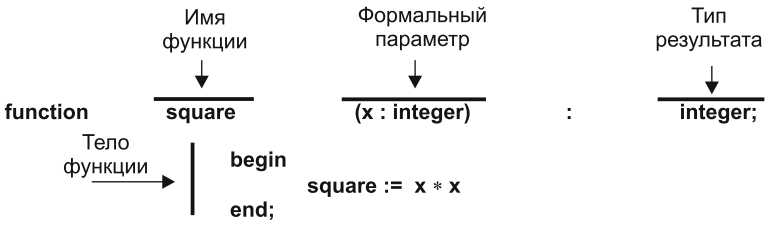


Рис. 13.2. Объявление функции square

Элементами объявления функции `square` являются:

- имя функции — `square`;
- формальный параметр — `x` типа `integer`;
- тип результата — `integer`;
- тело функции — возврат значения `x * x`.

Первые три элемента входят в состав заголовка функции. В языке Pascal заголовок функции начинается с ключевого слова `function`, а заголовок процедуры — со слова `procedure`. Заголовок часто называют *спецификацией*, а тело — *реализацией* функции.

Ту часть заголовка, которая задает типы формальных параметров, порядок их следования, а также тип данных результата, принято называть *сигнатурой* (или *прототипом*) функции.

На языке C спецификация какой-нибудь функции записывается в виде:

```
float Fun(float A, int B)
```

который определяет сигнатуру функции как

```
Fun : вещественное × целое → вещественное
```

Процедура на языке C синтаксически оформляется как функция, которая не возвращает результат, ее заголовок должен начинаться с ключевого слова `void`:

```
void Proc(float A, int B, float *C, int *D)
```

Здесь формальные параметры C и D определены как указатели.

Эта же спецификация в языке Ada более наглядно выражает возможные различия в смысле параметров:

```
procedure Proc (A: in float; B: in integer; C: in out float; D: out boolean)
```

Данный заголовок определяет процедуру со следующей сигнатурой:

```
Proc : вещественное1 × целое × вещественное2 → вещественное3 × булево значение
```

Ключевые слова in, out и in out указывают на направление передачи информации между формальным и фактическим параметром (in — передача из фактического в формальный параметр, out — из формального в фактический параметр, in out — двунаправленная передача).

Иначе говоря, через параметры A и B информация поступает в процедуру, через параметр C возможна двухсторонняя пересылка, а через параметр D информация только выводится из процедуры. Нетрудно догадаться, что здесь «обойден запрет» на возврат процедурой результата: фактически результаты работы процедуры поступают во внешнюю среду через параметры C и D (!).

При вызове функции выполняются операторы ее тела. В большинстве языков функция возвращает результат с помощью оператора return:

```
return x * x;
```

В языке Pascal для возвращения результата используется необычный синтаксис — возвращаемая величина присваивается имени функции. При этом в операторе тела имя функции рассматривается как переменная.

Заметим, что при попытке выяснить, какую математическую функцию вычисляет подпрограмма, возможны некоторые проблемы:

- в подпрограмме могут использоваться *неявные* параметры в виде ссылок на нелокальные переменные;
- в итоге работы подпрограммы могут появиться *неявные результаты* (*побочные эффекты*). Они возвращаются в виде изменения значений нелокальных переменных или в виде изменений значений ее параметров (категории in out);
- для некоторых значений параметров подпрограмма может быть не определена. В подобных ситуациях она не выполняет свои обычные вычисления, а завершается аварийно.

Работа подпрограммы определяется ее телом, которое состоит из объявлений локальных данных, описывающих используемые структуры данных, и операторов, задающих выполняемые действия. Объявления и операторы тела скрыты от пользователя. Пользователь может лишь вызвать подпрограмму с конкретным набором фактических параметров и получить вычисленные результаты. Синтаксис языка C и C++ для тела подпрограммы достаточно прост:

```
float Fun (float A, int B)  // спецификация подпрограммы
    {float M(10); int C;    // - объявления локальных объектов данных
    ...                     // последовательность операторов,
    }                       // определяющая действия подпрограммы
```

В некоторых языках (например, в языках Pascal и Ada, но не в C) в тело подпрограммы могут входить объявления других подпрограмм, работающих только в пределах объемлющей их подпрограммы. Эти локальные подпрограммы также скрыты, так что извне к ним обратиться нельзя.

Каждый вызов подпрограммы должен передать ей параметры правильного типа, в соответствии с ее спецификацией. Кроме того, требуется определить тип результата, возвращаемого подпрограммой. Контроль типов для подпрограмм аналогичен контролю типов для простых операций. Он может выполняться статически во время компиляции, если для типов фактических параметров и результатов всех подпрограмм определены объявления. Альтернативой считается динамический контроль типов, происходящий в период выполнения программы. Возможно и автоматическое (неявное) приведение параметра к правильному типу, если подобное действие разрешено реализацией языка. Специфика только в том, что в случае подпрограмм программист должен явно объявлять информацию о типах аргументов и результатов, в то время как для простых операций эта информация передается неявным образом.

Вызов подпрограммы

При вызове подпрограммы создается ее активация. Активации подпрограмм существуют только в период вычислений. После окончания выполнения подпрограммы активация разрушается. Если подпрограмма вызывается еще раз, создается новая активация. В ходе вычислений по единственному объявлению подпрограммы может быть создано много активаций. Их характеристики определяются содержанием объявления подпрограммы.

При создании активации ей должна быть выделена область памяти, которая освобождается при ее разрушении. Таким образом, период жизни активации — это промежуток времени между создающим ее вызовом подпрограммы и выходом из подпрограммы, когда активация разрушается.

В качестве примера рассмотрим объявление функции, написанное на языке C:

```
float Fun (float A, int B)
{
    const min = 10;
    #define max 70
    float M [20]; int C;
    C = min;
    if (C < max) {…}
    return (M [C] + 30 * A);
}
```

В нем определены все компоненты, необходимые для активации функции во время выполнения:

- Заголовок функции предоставляет информацию о памяти, необходимой для ее параметров (под объект данных типа `float` и объект данных типа `int`) и для вычисляемого ею результата (под объект данных типа `float`).
- Объявления тела определяют область памяти под локальные переменные — массив `M` и переменную `C`.
- Рассчитывается память под хранение литералов и именованных констант: `min` — это именованная константа со значением `10`, `max` — именованная константа со значением `70`, а `70` и `30` являются литералами.
- Определяется емкость памяти для исполняемого кода, который генерируется при компиляции операторов, содержащихся в теле подпрограммы.

В объявлении функции `Fun` предложение `const` сообщает компилятору, что объект данных `min` — это числовая константа со значением `10`. Строка же `#define max 70` является макрокомандой препроцессору, который преобразует каждое имя `max`, появившееся в тексте функции, в символ `70`. Следовательно, компилятор языка C даже не встретится с именем `max`, так как его везде заменит числовой литерал `70`. С точки зрения вычислений оба способа задания констант равнозначны, но смысл у предложений `const` и `#define` совершенно разный. Объект данных `min` обладает *l*-значением, чье *r*-значение равно `10`. У объекта `max` есть только *r*-значение, равное `70`.

Объявление подпрограммы позволяет еще в период компиляции организовать представление в памяти всех используемых объектов данных, а также определить исполняемый код.

Итогом компиляции становится шаблон, используемый для создания активации каждый раз, когда вызывается подпрограмма. Для рассматриваемой функции этот шаблон представлен на рис. 13.3.

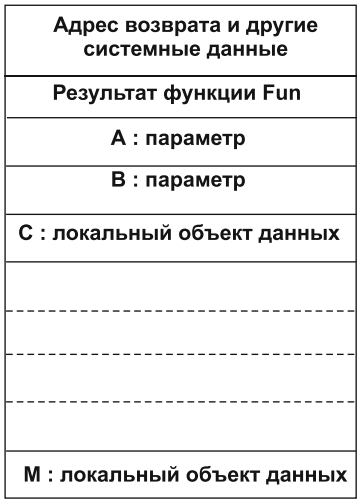


Рис. 13.3. Шаблон активации для функции Fun

Для создания конкретной активации функции на основе ее шаблона следует скопировать весь этот шаблон в новую область памяти. Однако вместо полного копирования целесообразнее разбить шаблон на две части, выделив в нем статическую и динамическую составляющие и расширив его за счет исполняемого кода подпрограммы.

Статическая составляющая называется *сегментом кода* и состоит из констант и выполняемого кода подпрограммы. Как показано на рис. 13.4, дополнительно сегмент включает в себя две секции:

1) код для создания записи активации;
2) код для удаления записи активации.

Когда вызывается подпрограмма, осуществляется ряд служебных действий, связанных с созданием записи активации, передачей параметров, формированием ссылок на нелокальные переменные и другими вспомогательными задачами. Эти действия должны быть произведены до того, как начнет выполняться код,

представляющий операторы тела подпрограммы. Поэтому вспомогательный *код инициализации* компилятор вставляет перед началом кода самой подпрограммы. По завершении выполнения подпрограммы тоже требуется осуществить ряд вспомогательных действий. Они связаны с возвращением результатов и освобождением памяти, выделенной под запись активации. Соответствующий *код финализации* вставляется в конец исполняемого кода подпрограммы.

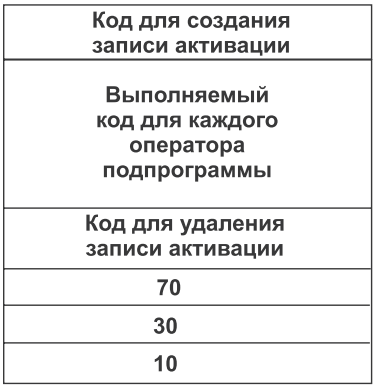


Рис. 13.4. Сегмент кода для функции Fun

Поскольку сегмент кода не меняется в ходе выполнения любой копии подпрограммы, он может быть использован повторно и для других активаций.

Динамическая составляющая шаблона называется *записью активации* и состоит из параметров, результатов функций и локальных данных, а также других вспомогательных данных, зависящих от реализации языка (временных областей памяти, адресов возврата из подпрограммы и связей для ссылок на нелокальные переменные). Структура записи активации также одинакова для всех активаций данной подпрограммы, но значения данных отличаются. Следовательно, для каждой активации требуется своя собственная запись активации.

Итоговую структуру кода для поддержки работы подпрограммы иллюстрирует рис. 13.5.

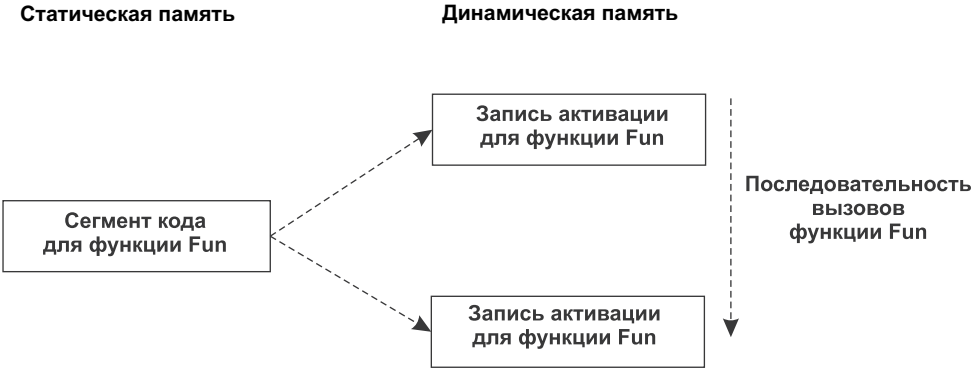


Рис. 13.5. Общий сегмент кода и различные записи активации для обеспечения вызовов подпрограмм

В период вычислений для каждой подпрограммы в памяти хранится только один сегмент кода. Записи активации динамически создаются при обращении к подпрограмме и уничтожаются каждый раз при завершении ее выполнения.

Управление памятью при вызове подпрограммы и выходе из нее сводится к выделению под запись активации блока памяти и его освобождению после завершения подпрограммы.

Рекурсия — множественные выполнения подпрограммы

Процедура (функция) *рекурсивна*, если она запускается из собственного тела (это может быть косвенный или прямой вызов). Рекурсия может быть запущена многократно.

Рассмотрим рекурсивную форму вычисления факториала:

```
function f (n : integer) : integer;
    begin
        if  n = 0  then  f := 1  else  f := n * f(n – 1)
    end;
```

Функция `f( )` является рекурсивной, так как `f(n)` вычисляется с помощью `f(n – 1)`, `f(n – 1)` — с помощью `f(n – 2)` и так далее до вычисления `f(0) = 1`.

Последовательность запусков при вызове функции `f(3)` отобразим графически:

```
f(3) = 3 * f(2) /* приостановка выполнения f(3) */
    f(2) = 2 * f(1) /* приостановка выполнения f(2) */
        f(1) = 1 * f(0) /* приостановка выполнения f(1) */
            f(0) = 1
        f(1) = 1 /* возобновление выполнения f(1) */
    f(2) = 2 /* возобновление выполнения f(2) */
f(3) = 6 /* возобновление выполнения f (3) */
```

В ответ на вызов `f(3)` тело функции выполняется с `n = 3`:

```
if  3 = 0  then  …  else …
```

Так как результат сравнения `3 = 0` равен `false`, начинает вычисляться выражение `3 * f(3 – 1) = 3 * f(2).`

Теперь для выполнения умножения требуется значение `f(2)`, поэтому для вычисления `f(2)` рекурсивно вызывается `f( )`. Аналогично в ходе запуска для `f(2)` вызывается `f(1)`. Далее, в ходе запуска `f(1)` вызывается `f(0)`.

Недостатком рекурсивных подпрограмм для обычных компьютеров является существенное увеличение расхода памяти, ведь под каждый вызов создается дополнительная активация подпрограммы. Хотя, конечно, математикам эта форма очень нравится — она соответствует естественной форме записи вычислений.

Преимущества подпрограмм

Подпрограмма — это «черный ящик»: известно «*что* она делает», а не «*как* она делает свою работу». Это отделение понятия «что» от понятия «как», или поведения

от реализации, позволяет разделить программу на части, каждая из которых может рассматриваться независимо.

К основным преимуществам подпрограмм следует отнести следующее:

1. *Уменьшение сложности программирования* (за счет повышения уровня абстракции). Использование только имени подпрограммы (`sort`) позволяет абстрагироваться от деталей реализации, думать в терминах операций, относящихся к решаемой проблеме.
2. *Закрытость реализации* — модификация алгоритма внутри подпрограммы не воздействует на остальную часть программы.
3. *Модульность программ* — подпрограммы используют для разбиения программы на небольшие куски, которые можно рассматривать отдельно. Поэтому подпрограммы позволяют нам управлять более крупными программами, чем те, которые мы можем мысленно обозревать.
4. *Расширение возможностей языков программирования* — создание библиотек (стандартных коллекций полезных подпрограмм). Операции типа + и * встраиваются в язык, но математические функции типа `sin( )`, `log( )` обеспечиваются библиотекой, как и процедуры ввода-вывода.

Методы передачи параметров

Формальные параметры в объявлении подпрограммы — это места для подстановки фактических параметров. Фактические параметры используют в вызовах подпрограмм. Например, в операторе вызова `F(2)` двойка является фактическим параметром. Рассмотрим формальный параметр `x` в следующем объявлении функции:

```
int square (int x)
{
    return x * x
};
```

Вызов `square(2)` возвращает `4`, вычисляя `x * x`, где вместо `x` подставляется `2`. Вызов `square(3)` возвращает `9`, вычисляя то же выражение `x * x` с `3` вместо `x`.

Методы передачи параметров — это способы, которыми параметры передаются в подпрограмму и/или возвращаются из нее.

Сопоставление между фактическими и формальными параметрами в вызове подпрограммы называют *передачей параметров*. Если параметров в подпрограмме несколько, то возникает необходимость корректного связывания соответствующих пар.

Опишем два способа, которые применяются для установления такого связывания.

Позиционное сопоставление. Соответствие между фактическими и формальными параметрами устанавливается на основе их позиций в списках фактических и формальных параметров: два параметра, которые занимают одинаковые позиции в списках, образуют пару. Таким образом, первый фактический и первый формальный параметр составляют пару, затем — вторые параметры из обоих списков и т. д.

Сопоставление по имени. В языке Ada и в некоторых других языках при вызове подпрограммы можно явно указать, какой формальный параметр должен соответствовать данному фактическому параметру. Например, в языке Ada можно вызвать подпрограмму `Sub` следующим образом:

```
Sub(X => A, Y => 27);
```

Здесь при вызове подпрограммы `Sub` фактический параметр `A` объединяется в пару с формальным параметром `X`, а фактический параметр `27` — с формальным параметром `Y`.

В языке Python функции тоже можно вызывать подобным способом:

```
adder (size = the_size,
    list = the_list,
    sum = the_sum)
```

Здесь `size`, `list` и `sum` — имена формальных параметров из объявления функции `adder`.

Главный недостаток сопоставления по имени заключается в том, что пользователь подпрограммы должен знать имена формальных параметров.

В большинстве языков используется исключительно позиционное сопоставление, поэтому в наших примерах также применяется этот способ. Обычно количество формальных и фактических параметров должно совпадать, чтобы соответствие между ними было взаимно однозначным. Тем не менее в некоторых языках это требование не выполняется, и в этом случае используются специальные соглашения, интерпретирующие отсутствие или избыток фактических параметров.

В языках Python, Ruby, Fortran 95 (и выше), PHP, C++ и Ada формальные параметры могут иметь значения по умолчанию. Значение по умолчанию используется, если формальному параметру, указанному в спецификации подпрограммы, не передается никакого фактического параметра.

Рассмотрим следующий заголовок функции на языке Python:

```
def salary(hours, tax_free = 1, hour_rate)
```

В этом языке все объявления функций начинаются с ключевого слова `def`. При вызове функции `salary` фактический параметр, соответствующий формальному параметру `tax_free`, можно пропустить. В этом случае вместо него используется значение `1` (значение по умолчанию). В вызовах функций на языке Python вместо пропущенного параметра запятая не ставится. Такая запятая должна указывать на позицию следующего параметра, а в этом нет никакой необходимости: все последующие фактические параметры должны сопоставляться по имени, как показано в следующем вызове функции:

```
my_money = salary(120, hour_rate = 10.0)
```

В языке C++ сопоставления по имени не предусмотрены, поэтому правила использования параметров по умолчанию совсем другие. Параметры по умолчанию должны указываться в конце списка, поскольку предусмотрено только позиционное сопоставление элементов списка. Если параметр со значением по умолчанию в вызове пропущен, то все последующие формальные параметры должны иметь значения по умолчанию. Заголовок функции `salary` на языке C++ следует записать иначе:

```
float salary (int hours, float hour_rate, int tax_free = 1)
```

Видим, что параметры в этой функции перечислены в другом порядке: параметр со значением по умолчанию записан последним. Теперь оператор вызова принимает следующий вид:

```
my_money = salary(120, 10.0);
```

Методы в языке C# могут иметь переменное количество параметров, если это параметры одного и того же типа. Подобные формальные параметры помечены модификатором `params`. В операторе вызова может быть записан массив или список выражений, чьи значения компилятор помещает в массив и предоставляет вызываемому методу. Обсудим следующий метод:

```
public void PutList(params int[] list) {
    foreach (int next in list) {
        Console.WriteLine("Next value {0}", next);
    }
}
```

Если метод `PutList` размещен в классе `TheClass` и имеются следующие объявления

```
TheClass theObject = new TheClass;
int[] littleList = new int[10] {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
```

то метод `PutList` можно будет вызвать любым из следующих операторов:

```
theObject.PutList(littleList);
theObject.PutList(5, 47 * a + 1, 114);
```

Язык Ruby предлагает сложную, но очень гибкую схему конфигурации фактических параметров. Здесь выделяют три категории фактических параметров, каждая из которых обслуживается соответствующим формальным параметром.

К первой категории относят выражения, чьи значения в виде объектов передаются в обычные формальные параметры.

Вторую категорию образует группа параметров, которые могут быть упакованы в список пар «ключ => значение». Этот список помещается в анонимный хеш, ссылка на который передается в единичный формальный параметр, называемый хеш-элементом. Данный механизм заменяет сопоставление по имени, которое непосредственно в Ruby не поддерживается.

Третью категорию представляют фактические параметры-массивы. Элементы массива могут иметь разные типы. Имя фактического параметра-массива должно предваряться звездочкой, например `*list`. Такому массиву сопоставляется специальный формальный параметр-массив с именем, которое тоже начинается со звездочки, например `*p4`. Параметр-массив должен быть последним в списке параметров метода. При вызове метода в формальный параметр-массив заносится ссылка на новый объект класса `Array`. Все оставшиеся фактические параметры присваиваются элементам данного объекта.

Таким образом, механизм обеспечения переменного количества параметров в Ruby похож на механизм, реализованный в C#. Поскольку массив в Ruby может хранить значения разных типов, здесь нет требования, чтобы фактические параметры, передаваемые через массив, были однотипными.

Проиллюстрируем механизм передачи параметров в какую-то функцию Ruby следующим примером:

```
list = [1, 2, 3, 4]
def funcR(p1, p2, p3, *p4)
. . .
end
. . .
funcR('teacher', growth => 175, weight => 70, age => 27, *list)
```

При выполнении этого оператора, который вызывает подпрограмму `funcR`, формальные параметры получат следующие значения:

`p1` хранит `'teacher'`.

`p2` хранит `{growth => 175, weight => 70, age => 27}`.

`p3` хранит `1`.

`p4` хранит `[2, 3, 4]`.

Передача параметров в языке Python похожа на схему, реализованную в Ruby.

Язык Lua использует простой механизм для поддержки переменного количества параметров — такие параметры представляются многоточием (. . .). Это многоточие рассматривается как массив, или список значений, которые могут быть присвоены списку переменных.

К примеру, рассмотрим две следующие функции:

```
function add (. . .)
    local sum = 0
    for i, next in ipairs{. . .} do
        sum = sum + next
    end
    return sum
end
```

Здесь `ipairs` является итератором для массивов (возвращает индекс и значение элементов массива, по одному элементу за раз). Фигурными скобками `{. . .}` обозначен массив фактических параметров.

```
function make (. . .)
    local a, b, c = . . .
. . .
end
```

Допустим, что оператор вызова функции `make` имеет вид:

```
make(5, 9, 2)
```

В этом случае `a`, `b` и `c` будут инициализированными в функции значениями `5`, `9` и `2` соответственно.

Параметру «три точки» не требуется быть единственным — он может появиться в конце списка именованных формальных параметров.

Вернемся к основной теме. Использование различных характеристик параметра приводит к различным методам передачи параметров. Например, при вызове процедуры `P(A[j])` возможны следующие варианты:

- Передача по значению. Передается значение `A[j]`.
- Передача по ссылке. Передается место размещения `A[j]`.
- Передача по имени. Передается сам текст `A[j]`.

Параметр, передаваемый по значению, называют *параметром-значением*.

Параметр, передаваемый по ссылке, называется *параметром-ссылкой*.

В языке Pascal параметры обычно передаются по значению. Если нужно передать данные по ссылке, то перед именем формального параметра указывается слово `var`. В C используется только передача по значению. В языке C++ применяют оба варианта.

Передача параметров по значению

При передаче по значению формальный параметр (*placeholder*) `x` подпрограммы `P(x)` получает значение фактического параметра. Иначе говоря, если параметр передается по значению, то *r*-значение фактического параметра передается формальному параметру вызванной подпрограммы.

При вызове `P(E)` используют следующую схему вычислений:

1. Для формального параметра `x` выделяется место во временной памяти подпрограммы.
2. Вычисляется значение фактического параметра (выражения `E`), оно заносится в ячейку формального параметра `x := E`.
3. Связь между формальным и фактическим параметром разрывается.
4. Выполняется тело подпрограммы, в котором используется формальный параметр.
5. Если `P(x)` — функция, то в точку вызова возвращается результат.

Следствия:

1. Фактический параметр может быть выражением.
2. С помощью формального параметра нельзя вернуть результат в точку вызова.

Вызов `square (E)` имеет следующий эффект.

Действие	Комментарий
x := E;	Передать значение E, присвоить его x
square := x * x	Выполнить тело функции `square(x)`
return square	Вернуть результат

Передача по значению используется чаще всего. В языках Pascal и C она рассматривается как основной метод передачи параметров. В этом случае формальный параметр содержит собственно значение фактического параметра (на момент вызова подпрограммы), которое и применяется для вычислений. Недостаток этого метода: итоги вычислений подпрограммы не отражаются на значениях фактических параметров. Любые изменения значения формального параметра, произошедшие во время выполнения подпрограммы, теряются, когда подпрограмма завершает свое выполнение.

Пример. Рассмотрим процедуру обмена двух переменных `swap(x, y)`, которая должна обеспечивать обмен значениями между `x` и `y`.

Эту процедуру будем использовать как индикатор, который «высвечивает» все особенности обсуждаемого метода передачи параметров. Используем передачу по значению:

```
procedure swap (x, y : T);
    var  z : T;
    begin
        z := x;  x := y;  y := z
    end;
```

Шаги вычислений при вызове `swap(a, b)` по значению имеют вид:

Действие	Комментарий
`x := a;`	Передача значения из `a` в `x`
`y := b;`	Передача значения из `b` в `y`
`z := x;  x := y;  y := z;`	`a` и `b` не изменяются

Видим, что, увы, обмен не состоялся. Несмотря на обмен значениями между формальными параметрами `x` и `y`, процедура не изменила значения фактических параметров `a` и `b`. Именно это обстоятельство и подвигнуло на придумывание дополнительных методов передачи параметров.

Передача параметров по ссылке

Передача по ссылке является, вероятно, наиболее распространенным механизмом передачи параметров. При *передаче по ссылке* формальный параметр превращается в синоним места размещения фактического параметра. Иными словами, передача объекта данных с помощью этого механизма означает, что подпрограмме становится доступным указатель на местоположение этого объекта (то есть его *l*-значение). При этом *расположение объекта данных* в памяти не изменяется. В начале выполнения подпрограммы *l*-значения всех фактических параметров используются для инициализации локальных областей памяти, отведенных под формальные параметры.

Передача параметров по ссылке осуществляется в два этапа:

1. В вызывающей программе определяются указатели на объекты данных, соответствующие фактическим параметрам (то есть их *l*-значения). Список таких указателей сохраняется в общей области памяти, доступной также и вызываемой подпрограмме (обычно этой областью служит набор регистров или программный стек). Затем управление передается вызываемой подпрограмме.
2. В вызванной подпрограмме список указателей на фактические параметры используется для вычислений, с целью получения требуемых *r*-значений этих параметров.

Во время выполнения подпрограммы ссылки на имена фактических параметров рассматриваются как ссылки на обычные локальные переменные. По окончании работы вызванной подпрограммы результаты возвращаются в вызывающую программу также через объекты данных, соответствующие фактическим параметрам.

Рассмотрим процедуру, в которой применяется и передача по ссылке, и передача по значению:

```
procedure P ( x : Tx; var  y : Ty ) … end;
```

Здесь для формального параметра `x` указана передача по значению. Напротив, для формального параметра `y` указана передача по ссылке (об этом говорит служебное слово `var`, поставленное перед именем параметра).

Вызов `P(a + b, c)` имеет следующий эффект:

Действие	Комментарий
`x := a + b`	Формальному параметру x присвоить значение суммы a + b
`py := &c` где py — служебный указатель подпрограммы, привязанный к формальному параметру y	Установить для y такое же место размещения, какое имеет c
Вычисления над x и *py	Выполнить тело процедуры P (обрабатываются формальный параметр x и фактический параметр c)

ПРИМЕЧАНИЕ

В этой таблице мы использовали обозначения языка C для операций взятия адреса и разыменования указателя.

Следствия:

1. Фактический *параметр-значение* может быть выражением, фактический *параметр-ссылка* выражением быть не может (так как у выражения нет места расположения), а должен быть именем переменной.
2. Место размещения фактического параметра-ссылки вычисляется и передается по ссылке до того, как начнет выполняться тело процедуры.

Изменим реализацию процедуры `swap` — используем для ее параметров только передачу по ссылке:

```
procedure swap ( var x : integer; var y : integer);
    z : integer;
    begin
        z := x;  x := y;  y := z;
    end;
```

Рассмотрим пример, где одним из фактических параметров является `A[`i`]` и значение i изменяется в процессе работы процедуры. Передается в процедуру то место размещения, которое элемент `A[`i`]` имеет на начало вызова. Так как и `x`, и `y` являются параметрами-ссылками, в ходе вычислений по вызову `swap(`i`, A[`i`])` выполняется следующее.

Действие	Комментарий
`px := &i`	Сделать место размещения x таким же, как у фактического параметра `i`
`py := &A[i]`	Сделать место размещения y таким же, как у фактического параметра `A[i]`
`z := *px;  *px := *py;  *py := z`	По сути действия реализуют следующие операторы: z := i; i := A[i]; A[i] := z

Таким образом, вычисления в теле процедуры производятся не над формальными, а над фактическими параметрами.

Кстати, обратите внимание, что адреса фактических параметров зафиксированы на момент вызова процедуры, а после этого не меняются. Именно поэтому значения *i* в левой и правой части оператора *i* := A[*i*] разные.

Фактически здесь указывается: занести в ячейку с именем *i* содержимое той ячейки массива A, индекс которой зафиксирован на момент вызова swap-процедуры.

Если i = 2, A[2] = 99, тогда последняя строка приведет к следующим действиям:

```
z := 2;     i := 99;      A[2] := z;
```

Вывод: обмен значениями *i* и A[*i*] происходит.

Эффект передачи параметров по ссылке с помощью указателей языка C

В языке C единственный метод передачи параметров — передача по значению. Эффект передачи по ссылке можно получить с помощью указателей.

Поскольку операция взятия адреса & в C создает указатель на переменную, то фактическими параметрами в вызове процедуры swapc(&a, &b) являются адреса переменных a и b.

Тело процедуры swapc() может менять значения a и b, используя косвенную адресацию, то есть разыменование.

Модифицируем процедуру swapc() следующим образом:

```
void  swapc  ( int *px,  int *py ) {
    int z;
    z = *px;   *px = *py;  *py = z;
}
```

В языке C префикс * является операцией разыменования указателя. Если pa — это указатель на a, то *pa можно поместить вместо a. Это означает:

- в правой части присваивания *pa обозначает значение a;
- в левой части присваивания *pa обозначает место размещения a.

Повторим, что здесь фактические параметры — адреса переменных, записанных в операторе вызова процедуры.

Вызов swapc(&a, &b) приводит к следующим действиям:

Действия	Комментарий
px = &a;	Передача адреса a в px
py = &b	Передача адреса b в py
z = *px;	Присваивание z начального значения a
*px = *py	Присваивание a значения переменной b
*py = z;	Присваивание b начального значения a, сохраненного в z

Вывод: обмен значениями между фактическими параметрами a и b происходит.

Достоинством метода передачи по ссылке является высокая эффективность с точки зрения как времени, так и памяти. Не требуется копировать значения фактических параметров, а затем отдельно обновлять их.

Обсудим недостатки передачи по ссылке:

1. Доступ к обрабатываемым параметрам замедляется, поскольку используется косвенная адресация.

2. Понижается безопасность хранения фактических параметров, поскольку двусторонний канал связи с ними открыт на все время работы подпрограммы. Могут возникнуть неумышленные и ошибочные изменения фактических параметров.
3. Появляется возможность возникновения псевдонимов (алиасов) фактических параметров. Известно, что псевдонимы снижают читабельность и надежность программ, а также усложняют их проверку.

Когда возникают псевдонимы? Чем чревато их появление? Рассмотрим эти вопросы подробнее.

Если имеется процедура на языке C с двумя параметрами-ссылками

```
void proc(int *px, int *py)
```

то вызов `proc(&arr[j], &arr[j])` приводит к тому, что `px` и `py` становятся алиасами элемента массива `arr[j]`. Следовательно, три имени `px`, `py` и `arr[j]` указывают на одну и ту же ячейку памяти. Последствия могут оказаться непредсказуемыми.

Кроме того, псевдонимы могут быть следствиями противоречий между формальными параметрами и глобальными переменными, которые становятся видимыми в подпрограмме. В качестве примера приведем следующий фрагмент на языке C:

```
int *global;
void main() {
    . . .
    proc(global);
    . . .
}
void proc(int *arg) {
. . .
}
```

Видим, что внутри процедуры `proc()` имена `arg` и `global` становятся псевдонимами (алиасами).

Передача по значению-результату

Второе название этого способа передачи параметров — «copy-in / copy-out».

В этом способе вначале фактические параметры копируются в формальные параметры, а в конце работы формальные параметры копируются обратно в фактические параметры.

Фактические параметры-выражения (например, 2+3), не имеющие места размещения, передаются по значению. Фактические параметры, имеющие место размещения, обрабатываются так:

1. *Этап copy-in.* Вычисляются значения и места размещения фактических параметров. Значения присваивают формальным параметрам. Места размещения сохраняют до этапа *copy-out*.
2. *Выполнение тела подпрограммы.* Здесь обрабатываются формальные параметры.
3. *Этап copy-out.* После выполнения тела подпрограммы конечные значения формальных параметров копируются обратно в места размещения, вычисленные на этапе *copy-in*.

В отличие от передачи по ссылке, здесь вся обработка внутри подпрограммы ведется в терминах формальных параметров. В момент вызова подпрограммы

значение фактического параметра (*r*-значение) копируется в объект данных, соответствующий формальному параметру, так что достигается такой же эффект, что и при явном присваивании значения фактического параметра формальному параметру. Во время выполнения подпрограммы каждое упоминание имени формального параметра соответствует обращению к локальной переменной подпрограммы. По завершении подпрограммы содержимое объекта данных, соответствующего формальному параметру, копируется в объект данных, обозначающий фактический параметр, моделируя явное присваивание финального значения формального параметра фактическому параметру. Таким образом, фактический параметр сохраняет свое значение вплоть до завершения подпрограммы, когда ему присваивается новое значение как результат работы подпрограммы.

Передача параметров по значению-результату была реализована Н. Виртом для языка Algol-W, который был преемником очень популярного языка Algol 60. На использованных им компьютерах отсутствовала возможность косвенного обращения к памяти. Поэтому передача параметров по значению-результату, делавшая все параметры локальными переменными (адресуемыми с помощью указателя текущей записи активации), позволяла ускорить вычисления.

При применении передачи по значению-результату оператор вызова `swap(`*i*`, A[`*i*`])` `swap`-процедуры по-прежнему обеспечивает обмен значений *i* и `A[`*i*`]`.

Опять допустим, что вначале *i* = 2, `A[2] = 99`. Так как места размещения фактических параметров вычислены на старте, на этапе *copy-out* рассматривается `A[2]`, хотя конечное значение *i* = `99`.

Передача по значению-результату отличается от передачи по ссылке только в случае алиасов (разных имен, обозначающих одно и то же место размещения).

Пример. Рассмотрим программу с алиасами:

```
program
. . .
procedure example (x, y); begin i := y end;
begin
    i := 2;   j := 3;
    example (i, j);
end.
```

Здесь процедура `example(x, y)` имеет два пути изменения значения переменной *i*:

- прямо, с помощью присваивания `i := y`;
- косвенно, с помощью этапа *copy-out* для параметра `x`.

Заметим, что косвенное изменение отменяет результат прямого присваивания.

В этой программе при вызове `example(i, j)` `i` передается как фактический параметр, поэтому формальный параметр `x` становится алиасом для `i`. Вызов `example(i, j)` оставляет `i` и `j` без изменений, поскольку этап *copy-out* восстанавливает их значения.

Проиллюстрируем отработку вызова `example(i, j)` следующей таблицей.

Действия	Комментарий
`px := &i;`	Сохранить место размещения фактического параметра `i`
`py := &j;`	Сохранить место размещения фактического параметра `j`
`x := i;`	copy-in значения фактического параметра `i` в формальный параметр `x`

Действия	Комментарий
`y := j;`	copy-in значения фактического параметра `j` в формальный параметр `y`
`i := y;`	Изменение значения `i`
`*px := x;` `*py := y;`	copy-out `x`, то есть восстановление `i`

Здесь `*px` и `*py` — косвенные адреса. Так как `px` хранит адрес `i`, присваивание *l*-значению `*px` имеет тот же эффект, что и занесение значения в переменную *i*. Аналогично присваивание *l*-значению `*py` имеет тот же эффект, что и занесение значения в переменную `j`. Таким образом, два последних присваивания восстанавливают значения `i` и `j`, сохраненные на этапе *copy-in*. С другой стороны, вызов по ссылке изменит значение `i`.

По сравнению с передачей по ссылке, передача по значению-результату дает следующие преимущества:

1. Доступ к обрабатываемым в подпрограмме параметрам ускоряется, поскольку в косвенной адресации нет необходимости.
2. Безопасность хранения фактических параметров повышается, так как канал связи с ними открывается на более короткое время (перед началом обработки в режиме приема и после окончания обработки в режиме записи). Во время выполнения тела подпрограммы канал закрыт.

Недостаток передачи по значению-результату состоит в необходимости хранить параметры в нескольких местах и тратить время на копирование их значений и адресов. Кроме того, здесь сохраняется проблема псевдонимов, хотя и проявляется она иначе.

Передача по результату

Параметр, передаваемый по результату, используется только для передачи результатов вычисления назад, в вызывающую программу. Начальное значение объекта данных, соответствующего фактическому параметру, не имеет никакого значения для подпрограммы и не может быть в ней использовано. Формальный параметр является локальной переменной (объектом данных), у которого начальное значение отсутствует (или инициализировано обычным способом, предусмотренным для локальных переменных). Когда выполнение подпрограммы завершается, конечное значение формального параметра присваивается фактическому параметру, как в случае передачи по значению-результату.

Одна из проблем при передаче по результату заключается в том, что может возникнуть конфликт на уровне фактических параметров, как в следующем примере:

```
proc (param, param);
```

Процедура `proc`, очевидно, имеет два формальных параметра с разными именами. Вероятно, что при ее выполнении эти параметры получат разные значения. Оба значения попадут в фактический параметр `param`. Возникает вопрос: в каком порядке будет происходить копирование? В данном случае именно порядок определит значение фактического параметра.

Допустим, что имеется следующий метод, написанный на языке C#:

```
void Report(out int a, out int b) {
    a = 29;
    b = 57;
}
```

В этом языке передача по результату задается спецификатором `out` при формальном параметре. При вызове

```
obj.Report(out m, out m);
```

возможны следующие ситуации:

- если первым присваивается значение формального параметра `a`, то значением фактического параметра вызывающего модуля станет `57`;
- если первым присваивается значение `b`, тогда значение фактического параметра будет равно `29`.

Поскольку порядок может задаваться реализацией, различные реализации языка будут генерировать разные результаты.

Другая проблема при передаче по результату состоит в выборе момента для вычисления адреса фактического параметра. Он может определяться в начальной фазе вызова подпрограммы или при возвращении из нее.

Предположим, что существует следующий метод, написанный на языке C#:

```
void Lottery(out int prize, out int position){
    prize = 780;
    …// Очень много операторов
    position = 77;
}
```

Далее вообразим, что где-то в вызывающем модуле записано:

```
. . .
index = 21;
obj.Lottery(myArray[index], index);
```

Адрес ячейки массива `myArray[index]` в промежутке между началом и завершением вызова метода `Lottery` изменяется. Если адрес определяется в начальной фазе вызова метода, то значение `780` будет занесено в ячейку `myArray[21]`. При вычислении адреса в момент возврата из метода число `780` сохраняется в ячейке `myArray[77]`. Очевидно, что неопределенность должен снять разработчик реализации языка.

Вызов процедуры с двумя идентичными параметрами может привести к проблемам и в других методах передачи параметров.

Правила области видимости для имен

В тексте программы имя может обозначать все, что угодно (переменную, константу, тип, процедуру,...). Чтобы что-нибудь сказать об имени, нужно посмотреть объявление (описание) имени:

```
var Z : T;        -- Z является переменной, имеющей тип T
```

Объявление имени задает его смысл.

Обработка имен в подпрограммах требует связывания между каждым использованием имени и его объявлением. Какое объявление имени x применимо к вхождению x в программу? Это определяют правила области видимости, иллюстрируемые рис. 13.6.

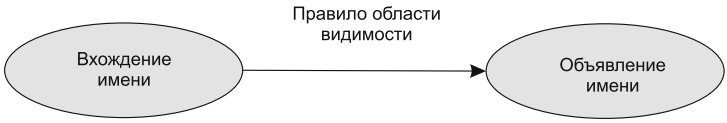


Рис. 13.6. Правило области видимости для имени

Базовая идея состоит в следующем: из точки, где появилось интересующее нас имя, должно быть видимо объявление, разъясняющее его смысл. Эту видимость должно обеспечивать *правило области видимости*. Правило области видимости очерчивает область, внутри которой объявление «обслуживает» любое вхождение некоторого имени.

Все эти рассуждения имеют прямое отношение к тем методам передачи параметров, которые эксплуатируют идею копирования тела подпрограммы в точку ее вызова и которые мы намерены обсудить.

В языках программирования различают статическую и динамическую области видимости.

Правила *статической области видимости* статически связывают вхождения имени с надлежащим объявлением, то есть в период компиляции.

Правила *динамической области видимости* определяют динамическое связывание вхождения имени с объявлением в период выполнения программы (период *runtime*). В этом случае искомым считается ближайшее (динамически) объявление.

Рассмотрим программу, в которой для одного вхождения имени предусмотрены два объявления. Программа содержит процедуру W, которая выводит значение некоторой переменной n:

```
procedure W; begin writeln (n); … end;
```

Переменная n не объявлена внутри W(), она объявляется в программе L или в другой процедуре D():

```
program L;
    var n : char;          {n объявлена в L}
    procedure W;
        begin
            writeln (n)    {вхождение n в W}
        end;
    procedure D;
        var n : char;      {n переобъявлена в D}
        begin
            n := ' D ';
            W              {W вызывается внутри D}
        end;
Begin          {начало L}
    n := ' L ';
    W;                     {W вызывается из главной программы L}
    D                      {D вызывается из главной программы L}
End.
```

Согласно статической области видимости корректное вхождение n в процедуре W( ) ограничивается объявлением в терминах исходного текста, анализируемого в период компиляции. При использовании правил языка Pascal или C вхождение n в W( ) относится к объявлению n в программе L, поэтому два вызова W( ) выведут на экран символы L и L.

При использовании правил традиционного языка LISP (в нем впервые применена динамическая область видимости) два вызова W( ) связывают n с разными объявлениями. В первом вызове (из тела основной программы) на экран выведется L. Во втором вызове — из процедуры D( ) — будет использовано локальное объявление n в процедуре D( ), поэтому на экран выведется D.

Чаще всего в императивных языках встречается статическая область видимости.

Статическая область видимости и переименование локальных переменных

Для устранения конфликтов имен достаточно последовательно переименовывать локальные переменные так, чтобы их имена не повторялись.

Для нашего примера следует изменить имя локальной переменной в процедуре D( ):

```
procedure D;
    var    r : char;
    begin
        r := 'D';
        W;
    end;
```

Логика вычислений не меняется, а источник конфликта устранен.

ПРИМЕЧАНИЕ

Конечно, это применимо, если не противоречит замыслу программиста. Мы полагаем, что совпадение имен носит случайный характер и программист не намеревался выводить значение локальной переменной из процедуры D( ) с помощью сторонней процедуры W( ). В противном случае надо переделывать процедуру W( ).

Принцип *переименования локальных переменных* гласит: каждое имя в программе должно иметь только одно объявление.

Макрорасширение и динамическая область видимости

Если тело подпрограммы просто копируется или подставляется в точку вызова, то мы получаем динамическую область видимости. Такая подстановка называется *макрорасширением*. Ее выполняет макропроцессор:

1. Вместо формальных параметров подставляется текст фактических параметров.
2. В точку вызова подставляется текст тела подпрограммы.

В языке C для поддержки расширений языка (введения именованных переменных и констант, подключения файлов) используется макропроцессор. Препроцессорные строки в C начинаются с символа #:

```
#define MaxBuf 4;
```

По этой директиве перед компиляцией программы каждое вхождение имени `MaxBuf` заменяется четверкой.

Конфликты именования

Текстуальная подстановка тела процедуры в точку вызова может привести к динамической области видимости. В этой области возможны конфликты между использованием имен в точке вызова и вызываемой подпрограмме.

Пример. Рассмотрим эффект макрорасширения вызова процедуры `W()` для программы `L`. Результат макроподстановки в процедуру `D()` имеет вид, представленный на рис. 13.7.

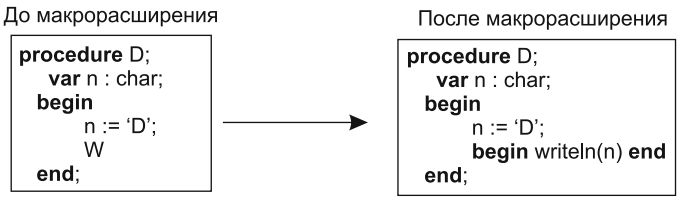


Рис. 13.7. Результат макроподстановки в процедуру D

Если тело процедуры `W()` подставляется в точку вызова в процедуре `D()`, то переменная `n` будет «покрываться» объявлением в процедуре `D()`. Теперь программа `L` выводит на экран символы `L` и `D` (появление эффекта динамической области видимости).

Передача параметров — текстуальная подстановка

Макрорасширение использует для передачи параметров текстуальную подстановку. Это может привести к результатам, отличным от любого другого метода передачи параметров.

Пример. Пусть при вызове процедуры обмена `swap(i, A[i])` выполняется макрорасширение.

Текстуальная подстановка фактических параметров в тело процедуры дает следующее:

```
z := i;    i := A[i];    A[i] := z;
```

Если вначале `i = 2`, `A[2] = 99`, то эта последовательность операторов будет эквивалентна:

```
Z := 2;    i := 99;   A[i] := z;
```

Видим, что конечное присваивание использует модифицированное значение `i`. Таким образом, начальное значение `i` заносится в `A[99]`, а не в `A[2]`. В результате обмен значениями не происходит.

Передача параметров по имени и статическая область видимости

В языке программирования Algol 60 имеется два механизма передачи параметров: *передача по имени*, *передача по значению* [82].

Правила для передачи по имени устраняют конфликты именования за счет переименования локальных переменных. Рассмотрим шаги устранения конфликтов.

Шаг 1. Вместо формальных параметров текстуально подставляются фактические параметры. В теле подпрограммы возможны конфликты имен (между именами в теле и именами фактических параметров). Конфликты устраняют переименованием локальных переменных тела процедуры.

Пример (Algol 60). Пусть оператором `P(A[i])` вызывается следующая процедура:

```
procedure P(x);
    begin
        integer i;
        …
        i := i + n;
        x := x + n;
    end;
```

Имеем: фактический параметр `A[i]` и локальную переменную `i`. Локальную переменную `i` в процедуре надо переименовать, например заменить ее имя на `j`. В результате подстановки и переименования получим модифицированное тело процедуры:

```
begin
    integer   j;
    …
    j := j + n;
    A[i] := A[i] + n;
end;
```

Шаг 2. Модифицированное тело процедуры подставляется в точку вызова. Возможны конфликты между нелокальными переменными в теле подпрограммы и локальными переменными в точке вызова. Их устраняют переименованием локальных переменных в точке вызова.

Положим, что `n` появляется как нелокальная переменная в теле процедуры. Пусть в точке вызова, находящейся в основной программе, `n` является локальной переменной:

```
Begin
    integer   n;
    n := n – 1;
    P(A[i]);
End.
```

В этом случае надо переименовать переменную `n` вызывающей программы. Выберем для нее имя `m`.

Тогда в результате переименования и подстановки получим следующий окончательный вариант вызывающей программы:

```
Begin
    integer  m;
    m := m – 1;
    begin
        integer  j;
        …
        j := j + n;
        A[i] := A[i] + n;
    end;
    …
End.
```

Таким образом, можно сделать вывод: передача параметров по имени имеет достаточные средства защиты от возможных конфликтов имен.

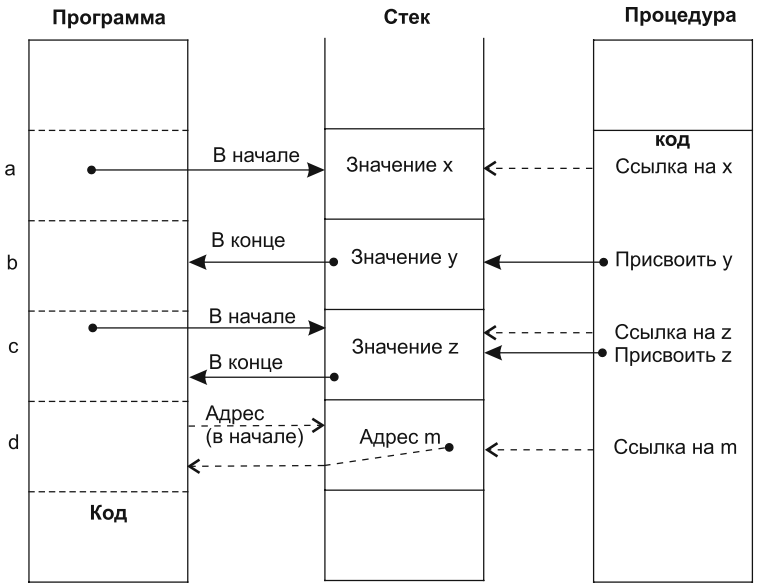
Реализация методов передачи параметров

В большинстве современных языков обмен параметрами происходит через стек выполняемой программы, иначе называемый стеком периода выполнения (runtime stack). Стек выполняемой программы инициализируется и поддерживается системой поддержки выполнения программ. Этот стек широко используется для управления подпрограммами и передачи параметров, как показано в главе 14. В ходе последующего обсуждения мы будем полагать, что стек используется для передачи всех параметров.

Параметры, передаваемые по значению, заносятся в ячейки стека. Далее эти ячейки хранят соответствующие формальные параметры. Логика передачи параметров по результату диаметрально противоположна. При передаче по результату значения, предназначенные для фактических параметров, помещаются в стек, откуда они извлекаются вызывающим программным модулем (после завершения работы вызванной подпрограммы). Передача параметров по значению-результату реализуется в полном соответствии со своей спецификой как комбинация передачи по значению и передачи по результату. Ячейка стека инициализируется вызовом и затем используется как локальная переменная в вызванной подпрограмме.

Передача параметров по ссылке наиболее проста для реализации. Независимо от типа фактического параметра в стек помещается лишь его адрес. Если параметр является литералом, в стек заносится адрес литерала. Если параметр является выражением, то компилятор создает программный код для вычисления выражения непосредственно перед передачей управления в вызываемую подпрограмму. Адрес ячейки памяти, в которую программный код заносит результат своих вычислений, затем записывается в стек. Компилятор должен гарантировать, что вызываемая подпрограмма не сможет изменить параметры, являющиеся литералами или выражениями.

Доступ к формальным параметрам в вызываемой подпрограмме осуществляется путем косвенной адресации по ячейке стека, хранящей нужный адрес. Передачи параметров по значению, по результату, по значению-результату, а также по ссылке, выполняемые на основе стека, иллюстрирует рис. 13.8.



Заголовок процедуры: void proc(int x, int y, int z, int m)
Вызов в программе: proc(a, b, c, d)
(передача: a — по значению, **b** — по результату, **c** — по значению-результату, **d** — по ссылке)

Рис. 13.8. Реализация методов передачи параметров на основе стека

Здесь показано, что из основной программы по оператору `proc(a, b, c, d)` вызывается процедура, для которой `a` передается по значению, `b` — по результату, `c` — по значению-результату, а `d` — по ссылке.

Методы передачи параметров в популярных языках программирования

В языке C используется передача параметров по значению. Передача по ссылке моделируется посредством применения указателей в качестве параметров. Значение указателя доступно вызываемой функции. Благодаря указателям вызываемая функция может изменять данные вызывающего модуля. Прообразом данного подхода послужил метод «передача по значению» языка ALGOL 68. Как в языке C, так и в языке C++ формальные параметры могут считаться указателями на константы. Соответствующим фактическим параметрам совсем не требуется быть константами. Такой подход обеспечивает эффективность передач по ссылке на основе семантики передачи по значению. Защита от записи параметров-указателей в вызываемой функции определена неявно.

Язык C++ содержит специальный тип указателей, называемых ссылками. Ссылки обсуждались в главе 11. Ссылки очень часто используются для передачи параметров в подпрограммы. Ссылочные параметры неявно разыменовываются, а их семантика полностью соответствует сути передачи по ссылке. В языке C++

ссылочные параметры можно определять как константы. Например, возможен следующий заголовок функции:

```
void sub(const int &p1, int p2, int &p3)
```

Здесь параметр p1 передается по ссылке, но не может быть изменен функцией sub, параметр p2 передается по значению, а параметр p3 — тоже по ссылке (но не является константой). Ни для параметра p1, ни для параметра p3 не требуется явное разыменовывание в функции sub.

В качестве примера перепишем с помощью ссылок swapc-процедуру, рассмотренную в разделе «Передача параметров по ссылке»:

```
void swapcpp(int &x, int &y) {
    int z;
    z = x;
    x = y;
    y = z;
}
```

Как видим, текст упростился: исчезли все признаки явного разыменования формальных параметров x и y — в этом нет необходимости, поскольку теперь они стали параметрами-ссылками, которые всегда неявно разыменованы.

Процедуру можно вызвать следующим образом:

```
swapcpp(a, b);
```

Параметры, являющиеся константами, несколько отличаются от входных параметров, для которых задан вид (режим) in. Параметры-константы очевидным образом поддерживают режим ввода данных в подпрограмму. Однако во всех широко распространенных императивных языках, за исключением языка Ada, значения входных формальных параметров в подпрограмме можно изменять. Правда, эти изменения никак не повлияют на значения соответствующих фактических параметров. Параметрам-константами никогда и ничего присваивать нельзя.

В языке Java все параметры как бы передаются по значению, хотя нужно сделать оговорку. Доступ к объектам в этом языке разрешен только с помощью ссылок. Объекты, являющиеся параметрами, в действительности передаются по ссылке. Хотя параметр-ссылку на объект в вызываемой подпрограмме изменить нельзя, сам объект может подвергаться изменениям (при наличии у него соответствующих методов). Ссылки не могут прямо указывать на скалярные переменные, поэтому скалярные переменные не передаются по ссылке. И тем не менее они доступны — через объекты, хранящие скалярные переменные.

В языках Ada и Fortran 95 для каждого формального параметра может быть определен режим in, out или inout.

В качестве стандартного метода в языке C# принят метод передачи по значению. Передача по ссылке помечается спецификатором ref, который предваряет формальный параметр и соответствующий ему фактический параметр. Например, рассмотрим следующий фрагмент на языке C#:

```
void adder(ref int total, int fee) { . . . }
. . .
adder(ref count, additive);
```

Первый параметр передается в операцию adder по ссылке, а второй — по значению.

Кроме того, C# поддерживает «выходные» параметры, которые передаются по результату и не имеют начальных значений. В списке параметров такие параметры имеют пометку `out`.

Передачи параметров в PHP такие же, как и в C#, однако для определения передачи по ссылке помечается или фактический, или формальный параметр. Имя такого параметра должно начинаться со знака амперсанда.

Язык Perl использует простейшие средства для передачи параметров. Все фактические параметры неявно помещаются в предопределенный массив с именем `@_`. Подпрограмма получает значения (или адреса) фактических параметров из этого массива. Наиболее удивительно то, что все элементы массива являются псевдонимами (алиасами) для фактических параметров. Поэтому, если элемент из массива `@_` изменяется вызванной подпрограммой, то это изменение отражается в соответствующем фактическом параметре точки вызова, если таковой параметр имеется и является переменной. Количество фактических параметров может отличаться от количества формальных параметров.

Метод передачи параметров в языках Python и Ruby называется *передачей по присваиванию*. Поскольку все элементы данных здесь считаются объектами, каждая переменная является ссылкой на объект. При передаче по присваиванию значение фактического параметра присваивается формальному параметру. С одной стороны, передача по присваиванию аналогична передаче по ссылке, поскольку значениями всех фактических параметров являются ссылки. С другой стороны, аналогия уместна лишь в определенных случаях. Например, многие объекты являются неизменяемыми.

В чисто объектно-ориентированном языке оператор присваивания `a = a + 1` не меняет объект, на который указывает ссылка `a`. Здесь подразумевается несколько иная последовательность действий: принимается объект, адресуемый с помощью `a`, объект увеличивается на 1, в результате чего создается новый объект со значением `a+1`, адрес нового объекта заносится в `a`. Поэтому, когда в подпрограмму передается ссылка на скалярный объект, сам адресат не может быть изменен. Поскольку ссылка передается по значению, изменение формального параметра в подпрограмме не влияет на фактический параметр в вызывающем модуле.

Теперь допустим, что в качестве параметра передается ссылка на массив. Если соответствующему формальному параметру присваивается новый объект-массив, это никак не влияет на вызывающий модуль. Однако если формальный параметр используется для присваивания значения элементу массива `list[7] = 54`, то фактический параметр изменяется. Следовательно, изменение ссылки формального параметра не влияет на вызывающий модуль, а на изменение элемента массива, который передается как параметр, вызывающий модуль реагирует.

Подводя итог, отметим, что метод передачи параметров по присваиванию сильно отличается от передачи параметров по ссылке в языке C++, но весьма напоминает передачу параметров в языке C:

- *Неизменяемые объекты передаются «по значению»*. Такие объекты, как целые числа и строки, передаются в виде ссылок на объекты, а не в виде копий объектов, но так как неизменяемые объекты невозможно изменить непосредственно, передача таких объектов похожа на копирование.

- *Изменяемые объекты передаются «по указателю»*. Такие объекты, как массивы (списки) и словари, также передаются в виде ссылок на объекты, что очень похоже на передачу указателей на массивы в языке C, — изменяемые объекты допускают возможность непосредственного изменения внутри подпрограммы так же, как и массивы в языке C.

В целом же, при передаче по присваиванию выполняется присваивание ссылок на изменяемые и неизменяемые объекты формальным параметрам подпрограммы.

Проверка типов параметров

Общепризнано, что требования надежности диктуют: типы фактических параметров должны проверяться на совместимость с типами соответствующих формальных параметров. Без этой проверки любая опечатка может привести к трудно обнаруживаемой ошибке вычислений. Рассмотрим следующий вызов функции:

```
total = func(47);
```

Здесь фактический параметр является целочисленной константой. Допустим, что формальный параметр функции `func` имеет тип `float`. Такое несоответствие можно выявить только при проверке типов. При наличии подобной ошибки функция `func` не сможет вычислить правильный результат.

Ранние языки программирования, такие как Fortran 77 и начальная версия C, не требовали проверки типов, в то время как более поздние языки предусматривали ее. Тем не менее относительно новые языки Perl, JavaScript и PHP тоже отказались от проверки типов.

В оригинальной версии языка C ни количество, ни типы параметров не проверялись. В стандарте языка C (1999 год) и языке C++ для формальных параметров функций стало обязательным наличие прототипа. Прототип требовал перечислять в скобках типы и имена параметров:

```
long myfun(long x)
{ . . . }
```

При наличии следующего вызова функции `myfun`:

```
value = myfun(arg);
```

тип фактического параметра, например `short`, сравнивался с типом формального параметра (`long`). При их несовпадении тип `short` неявно приводился к типу `long` (расширяющее приведение). Если приведение типов оказывалось невозможным или количество параметров не совпадало, то фиксировалась семантическая ошибка.

Однако проверки типов некоторых параметров можно избежать, заменив последнюю часть списка параметров многоточием:

```
int printf(const char *my_string, . . .);
```

Вызов функции `printf` должен иметь хотя бы один параметр, указывающий на строку из символьных литералов. Все остальное (включая отсутствие других параметров) считается вполне допустимым. Способ, которым функция `printf` определяет, есть ли в вызове дополнительные параметры, основывается на использовании спецификаторов форматирования в параметре, являющемся строкой.

Например, спецификатор форматирования для вывода целого числа имеет вид `%d`. Он является частью строки, как показано в следующем примере:

```
printf("The total is %d\n", total);
```

Символ `%` сообщает, что в функции `printf` более одного параметра.

Достаточно интересная ситуация возникает в C# при приведении фактического параметра к формальному, если по ссылке передается элементарная величина. Допустим, величина типа `float` передается в формальный параметр типа `double`. Если этот параметр передается по значению, то выполняется расширяющее приведение и никаких проблем не возникает. Подобное неявное приведение достаточно полезно, поскольку позволяет иметь в библиотеке подпрограммы над переменными типа `double`, которые можно использовать с величинами как типа `float`, так и типа `double`. Теперь предположим, что параметр передается по ссылке. Когда значение формального параметра типа `double` возвратится в фактический параметр типа `float`, возникнет переполнение в вызывающем модуле. Чтобы избежать этой проблемы в C#, требуется точное совпадение типа для пары «ссылочный фактический параметр + ссылочный формальный параметр». В этом случае неявное приведение не потребуется.

В языках Python и Ruby проверка типов параметров не предусмотрена, поскольку концепция типизации в этих языках совсем другая. У объектов есть типы, а у переменных они отсутствуют, поэтому формальные параметры лишены типов. Следовательно, проверка типов параметров становится невозможна в принципе.

Массивы в качестве параметров

В некоторых языках, например C и C++, при передаче многомерного массива как параметра в подпрограмму компилятор должен создавать адресную формулу для этого массива прямо по тексту подпрограммы, поскольку подпрограммы могут компилироваться отдельно от программ, их вызывающих. Рассмотрим специфику передачи матрицы в функцию на языке C. Многомерные массивы в языке C представляются как массивы массивов и хранятся построчно. Вспомним, что нижняя граница индексов здесь равна 0. Тогда адресная формула для возрастающего порядка строк матрицы и единичной длины элементов принимает вид:

```
адрес(matr[i, j]) = адрес(matr[0,0]) + i * количество_столбцов + j
```

Заметим, что учитывается количество столбцов, а не количество строк. Следовательно, в языках C и C++ при передаче матрицы в качестве параметра формальный параметр должен содержать количество столбцов, указанное во второй паре квадратных скобок. Это иллюстрируется следующим фрагментом на языке C:

```
void sub(int matrix[][50]) {
. . . }
void main() {
    int matr[10][50];
    . . .
    sub(matr);
. . .
}
```

Увы, но такой способ передачи не позволяет написать функцию, которая способна получать матрицу с разными количествами столбцов; для каждой новой матрицы с другим количеством столбцов нужно создавать новую функцию. Следовательно, вносится запрет на гибкие функции, которые можно было бы многократно применять к многомерным массивам. Запрет можно обойти при использовании арифметики указателей: матрица передается как указатель, а ее фактические размеры передаются как параметры. Тогда функция может вычислять адресную формулу, определенную пользователем, с помощью арифметики указателей каждый раз, когда нужно обратиться к элементу матрицы. Обсудим следующий прототип функции:

```
void sub(float *ptr_matr,
    int num_rows,
    int num_cols);
```

Для присвоения значения переменной `a` элементу `[row][col]` из матрицы, являющейся параметром подпрограммы `sub`, можно воспользоваться следующим оператором:

```
*(ptr_matr + (row * num_cols) + col) = a;
```

Конечно, этот оператор работает, но по форме он сложен. Наглядность можно повысить, написав макрос для определения адреса, например:

```
#define matr(r,c) (*(ptr_matr + ((r) * (num_cols) + (c)))
```

С учетом макроса оператор, записанный выше, можно модифицировать следующим образом:

```
matr(row,col) = a;
```

В других языках для передачи многомерных массивов в качестве параметров используются иные решения. Компиляторы языка Ada способны определять указанные размеры всех массивов, передаваемых как параметры, во время компиляции подпрограмм. В языке Ada формальными параметрами могут быть неограниченные массивы. Неограниченным называют такой массив, в котором диапазон изменения индексов в объявлении массива не задается. Диапазоны изменения индексов должны содержаться в объявлениях переменных, принадлежащих к типу «неограниченный массив». Подпрограмма, в которую неограниченный массив передается как фактический параметр, получает сведения о диапазоне изменения индексов в самом параметре. Допустим, что имеются следующие объявления:

```
type matrix is array (integer range <>, integer range <>) of float;
matr : matrix(1..400, 1..100);
```

Функция, которая возвращает сумму всех элементов массивов типа `matrix`, может быть объявлена следующим образом:

```
function adder (fm : in matrix) return float is
    rez : float := 0.0;
    begin
        for i in fm'range(1) loop
            for j in fm'range(2) loop
                rez:= rez + fm(i, j);
            end loop; -- цикл для j
        end loop; -- цикл для i
        return rez;
    end adder;
```

Атрибутная функция `range` возвращает диапазон изменения указанного индекса массива (номер индекса указан в круглых скобках), являющегося фактическим параметром. Заметим, что этот диапазон вполне конкретен, ведь фактический параметр является ограниченным двумерным массивом. В языке Ada любая атрибутная функция отделяется апострофом от того объекта (или типа), характеристику которого она определяет.

В языках Java и C# для передачи многомерных массивов в качестве параметров используется способ, похожий на способ, принятый в языке Ada. В языках Java и C# массивы являются объектами. Все они одномерны, однако их элементы могут быть массивами. Иными словами, здесь матрицы являются массивами массивов. Каждый массив наследует именованную константу (`length` в Java и `Length` в C#), определяющую длину массива при создании соответствующего объекта. Формальный параметр для матрицы записывается с двумя парами квадратных скобок, как показано в объявлении метода Java, являющегося полным аналогом рассмотренной функции на языке Ada:

```
float adder (float fm[][]) {
    float rez = 0.0f;
    for (int i = 0; i < fm.length; i++) {
        for (int j = 0; j < fm[i].length; j++) {
            rez += fm[i][j];
        } //** цикл для j
    } //** цикл для i
    return rez;
}
```

Поскольку каждый одномерный массив характеризуется своей собственной длиной, длина одной строки в матрице вполне может отличаться от длины другой строки.

Подпрограммы в качестве параметров

В вычислительных задачах достаточно часто возникают ситуации, когда внутри какой-то функции приходится задействовать одну из набора иных, вспомогательных функций. Это приводит к необходимости передач имен одних подпрограмм как параметров в другие подпрограммы. Например, в подпрограмме расчета эффективности применяются подпрограммы оценки частных показателей качества, в подпрограммах численного интегрирования — подпрограммы вычисления конкретной функции и т. д.

Реализация этой простой идеи сталкивается с некоторыми проблемами. Их пояснению посвящены два следующих подраздела.

Проверка типов параметров при вызовах подпрограммы

Первая проблема связана с проверкой типов параметров при вызовах подпрограммы, переданной как параметр.

Языки C и C++ в качестве параметров разрешают передавать указатели на функции. Типом указателя на функцию является ее протокол. Протокол функции — это количество, порядок и типы формальных параметров, а также тип возвращаемого ею значения. Поскольку протокол содержит типы всех параметров, такие параметры могут быть полностью проверены на совместимость типов.

К примеру, следующее объявление определяет указатель `p_func`, который может указывать любую функцию с параметрами типа `float` и `int` и возвращаемым результатом типа `float`:

```
float (*p_func)(float, int);
```

Любая функция с протоколом, обозначенным в этом указателе, может использоваться для инициализации данного указателя или назначаться указателю в программе. В C и C++ имя функции без последующих круглых скобок, как и имя массива без квадратных скобок, считается адресом функции (или массива). Следовательно, допустимы следующие способы инициализации или присваивания адреса указателю на функцию:

```
int my_func2 (int, int);              // Объявление функции
int (*p_func2)(int, int) = my_func2; // Создание указателя
                                      // и его инициализация для my_func2
p_func2 = my_func2;                   // Присваивание адреса функции указателю
```

Функцию `my_func2` теперь можно вызвать любым из следующих операторов:

```
(*p_func2)(a, b);
p_func2(a, b);
```

Первый оператор явно разыменовывает указатель `p_func2`, это корректно, но не обязательно.

Функциональные указатели в C и C++ могут передаваться в подпрограммы как параметры, а также использоваться для приема возвращаемых результатов, хотя сами функции могут и не применяться непосредственно в одной из этих ролей.

В языке C# мощь и гибкость указателей на методы усилилась благодаря превращению подобных указателей в объекты. Они называются *делегатами*, поскольку вместо вызова метода программа делегирует это действие делегату.

Для использования делегата предварительно нужно определить класс делегата. Этот класс должен зафиксировать протокол конкретного метода. Экземпляр делегата содержит имя метода с протоколом делегата, обеспечивающим вызов. Синтаксис у объявления делегата подобен объявлению метода, за исключением того, что перед именем типа результата записывается зарезервированное слово `delegate`. К примеру, возможно следующее объявление:

```
public delegate int Builder(int x);
```

Этот делегат может конкретизироваться любым методом с параметром типа `int` и возвращаемым результатом типа `int`. Например, рассмотрим следующее объявление метода:

```
static int func3(int a);
```

Делегат `builder` может инициироваться передачей конструктору делегата имени этого метода:

```
Builder myBuilder = new Builder(func3);
```

Это можно записать короче:

```
Builder myBuilder = func3;
```

Возможен следующий вызов метода `func3` с помощью делегата `myBuilder`:

```
myBuilder(47);
```

Возвращаясь к основной теме подраздела, отметим, что языки Fortran 95 и Ada также имеют механизмы для предоставления информации о типах параметров и результата тех подпрограмм, которые передаются в качестве параметров. Здесь тоже обеспечивается их проверка.

Организация области видимости для выполнения переданной подпрограммы

Вторая проблема проявляется в языках, где разрешены вложенные подпрограммы. Вопрос заключается в организации области видимости для выполнения переданной подпрограммы. Существуют три возможности:

- Область видимости ограничивается областью видимости для того оператора вызова, который активирует переданную подпрограмму (теневое связывание).
- Область видимости образует программный фрагмент, в котором объявлена переданная подпрограмма (глубокое связывание).
- Область видимости ограничивается областью видимости того оператора вызова, который передал подпрограмму как фактический параметр (специальное связывание).

Следующий фрагмент, представленный в синтаксисе языка JavaScript, иллюстрирует все эти возможности.

```
function func1() {
    var rez;
    function func2() {
        alert(rez); // создает диалоговое окно со значением rez
    };
    function func3() {
        var rez;
        rez = 3;
        func4(func2);
    };
    function func4(sub) {
        var rez;
        x = 4;
        sub();
        };
    rez = 1;
    func3();
};
```

Здесь в объявление функции `func1` вложены объявления функций `func2`, `func3` и `func4`. В заголовке функции `func4` обозначен формальный параметр-функция `sub`. При вызове из функции `func3` происходит передача имени функции `func2` в качестве параметра: фактический параметр `func2` заменяет формальный параметр `sub`. Работа фрагмента начинается с выполнения операторов тела функции `func1`:

```
rez = 1;
func3();
```

Далее вызывается функция `func3`, которая, в свою очередь, вызывает функцию `func4`. Рассмотрим выполнение функции `func2` при ее вызове из `func4`. При теневом связывании область видимости такого выполнения совпадает с областью видимости функции `func4`, так что вхождение переменной `rez` в функции `func2` связывается с локальной переменной `rez` в функции `func4`, и программа выводит строку `rez = 4`. При глубоком связывании область видимости выполнения функции `func2` совпадает с областью видимости функции `func1`, поэтому вхождение переменной `rez` в функции `func2` связывается с локальной переменной `rez` в функции `func1`, и программа выводит строку `rez = 1`. При специальном связывании вхождение переменной `rez` в функции `func2` связывается с локальной переменной `rez` в функции `func3`, и программа выводит строку `rez = 3`.

В некоторых случаях подпрограмма, в которую передается другая подпрограмма, также передает ее как параметр. В этих случаях глубокое связывание и специальное связывание совпадают друг с другом. Специальное связывание практически никогда не использовалось, так как можно предположить, что среда, в которой процедура оказывается в качестве параметра, не имеет естественной связи с переданной подпрограммой.

Теневое связывание неприемлемо для тех языков со статической областью видимости, где разрешены вложенные подпрограммы. Допустим, что процедура `Отправитель` передает процедуру `Посылка` как параметр в процедуру `Получатель`. Проблема заключается в том, что процедура `Получатель` может не принадлежать к статической области видимости процедуры `Посылка`, делая таким образом очень неестественным для процедуры `Посылка` доступ к переменным процедуры `Получатель`. С другой стороны, для любой процедуры в языках со статической областью видимости, включая процедуры, передаваемые как параметры, вполне нормальна ситуация, когда область видимости определяется местонахождением ее объявления. Следовательно, для языков со статической областью видимости более логичным является использование глубокого связывания. Некоторые языки с динамической областью видимости используют теневое связывание.

Типы возвращаемых значений

Большинство императивных языков программирования ограничивают типы значений, которые могут возвращаться их функциями. Язык C разрешает возвращать из функции значение любого типа, за исключением массивов и функций. Массивы и функции могут возвращаться с применением указателей. Язык C++ подобен C, но позволяет возвращать из функций значения типов, определенных пользователем.

Ada, Python, Ruby и Lua являются единственными среди императивных языков, чьи функции (и/или методы) могут возвращать значения любого типа. Правда, в языке Ada функции не являются типами, поэтому не могут возвращаться из функций. Конечно, указатели на функции возвращать можно.

В некоторых языках программирования подпрограммы считаются объектами «первого класса». Это значит, что они могут передаваться как параметры, возвра-

щаться из функций и присваиваться переменным. В некоторых императивных языках (например, в языках Python, Ruby и Lua) методы тоже считаются объектами «первого класса». Все это справедливо для функций в большинстве функциональных языков.

В языках Java и C# функции отсутствуют, хотя их методы подобны функциям. Из методов разрешается возвращать значения любого типа или класса. Поскольку методы не являются типами, они возвращаться не могут.

Количество возвращаемых значений

В большинстве языков из функции можно вернуть только одно значение, но есть и исключения. Язык Ruby позволяет вернуть из метода более одного значения. Если в методе Ruby оператор `return` не содержит выражения, то возвращается `nil`. Если же указано одно выражение, то возвращается его значение. В случае нескольких выражений возвращается массив из значений всех выражений.

Язык Lua также позволяет функциям возвращать множество значений. Такие величины записываются в операторе `return` в виде списка, элементы которого отделяются запятыми:

```
return 47, a, b, rez
```

Форма оператора вызова определяет количество величин, которые возвращаются вызывающему модулю. Если функция вызывается как процедура, то (по определению) все возвращаемые величины игнорируются. Если функция возвращает три значения и все они должны сохраняться вызывающим модулем, то оператор вызова принимает следующий вид:

```
x, y, z = func()
```

В языке F# для возврата множества значений их нужно разместить в кортеже, обеспечив, чтобы этот кортеж был последним выражением в функции.

Побочные эффекты функций

В главе 8 упоминалась возможность побочных эффектов при выполнении функций. Для их исключения желательно, чтобы параметры функций были входными, то есть неизменяемыми. Это требование эффективно защищает функции от сторонних эффектов через их параметры, псевдонимы параметров и глобальные переменные. Долгое время этот бастион защиты держал язык Ada, разрешавший для формальных параметров функций только режим «in». Но в версии 2012 года защита, увы, пала и это ограничение было снято. В большинстве других императивных языков разрешены параметры функций с передачей по ссылке, позволяя функциям подвергаться многочисленным побочным эффектам, а также открывая дорогу псевдонимам.

Чисто функциональные языки, такие как Haskell, не имеют переменных, поэтому их функции не могут иметь побочных эффектов.

Полиморфизм в языках программирования

Слово полиморфизм древнегреческого происхождения и обозначает примерно «много форм» (poly — много, morphos — форма). Слово morphos ассоциируется

с богом сна Морфеем (Morphus), который мог представляться спящим людям в разных обличиях и, следовательно, был действительно полиморфным.

В языках программирования под полиморфизмом понимается возможность писать программный код исходя из абстрактного интерфейса и детализировать его теми или иными конкретными реализациями. Интерфейс — это часть программного модуля, видимая для его клиентов. Механизмы, посредством которых в языках программирования обеспечивается поддержка полиморфизма, очень разнообразны. Согласно классическому определению, сформулированному Стречи в 1967 году, есть два вида полиморфизма: параметрический (parametric polymorphism) и специальный (ad hoc polymorphism) [88].

«Параметрический полиморфизм имеет место тогда, когда какая-либо функция действует единообразно в отношении множества типов; как правило, у этих типов общая структура. Специальным полиморфизмом называется ситуация, при которой поведение функции в отношении каждого отдельного типа (из множества типов с разнородными структурами) может различаться».

В дальнейшем Карделли и Вегнер развили это определение, выделив универсальный и специальный полиморфизм [56]. К универсальному они отнесли параметрический полиморфизм и полиморфизм включением, а к специальному — перегрузку и неявное приведение.

Различают полиморфные объекты и полиморфные функции. Полиморфный объект — это сущность (переменная, аргумент функции), хранящая во время выполнения программы значения различных типов. Полиморфные функции — это те функции, которые имеют полиморфные аргументы.

В общем случае полиморфизм в языках программирования отражает принцип подстановки. То есть полиморфной переменной разрешено хранить в себе значение, относящееся к ее объявленному типу данных или же к любому его подтипу.

Параметрический полиморфизм

Параметрический полиморфизм характеризуется применением родовых параметров. Основная функция родовых параметров заключается в том, чтобы минимизировать дублирование программного кода в языках со статическим контролем типов. Приведем пример. Допустим, что нам нужна функция возведения числа в квадрат. Псевдокод этой функции может иметь следующий вид:

```
sqr(x)
    return x * x
```

На языке с динамическим контролем типов наподобие Smalltalk этот псевдокод можно воспроизвести практически без изменений:

```
sqr: x
    ^x * x
```

Если же мы собираемся выполнить реализацию на языке со статическим контролем типов, необходимо объявить тип `x` и тип результата. В C++ это делается следующим образом:

```
int sqr(int x)
{ return x * x; }
```

Такая реализация справедлива только в отношении переменной `x` типа `int`, а значит, она более специализирована, чем псевдокод или код на Smalltalk, который

применим по отношению к любому объекту, способному истолковать сообщение «*» — другими словами, к любому типу, в котором есть операция «*». Мы могли бы написать на C++ несколько функций возведения в квадрат чисел различных типов — например, таких:

```
int sqrInt(int x)
{ return x * x; }

double sqrDouble(double x)
{ return x * x; }
```

Наилучшее решение предполагает использование типов в качестве параметров. Родовые параметры в C++ поддерживаются в виде шаблонов функций и классов. Следовательно, функцию возведения в квадрат можно реализовать в качестве шаблона функции:

```
template <typename T>
T sqr(T x)
{ return x * x; }
```

В результате для каждого типа параметра, с которым будет вызываться функция, компилятор будет автоматически генерировать соответствующую конкретную функцию — к примеру:

```
int a = 3;
double b = 5.0;
int x = sqr(a);    // компилятор генерирует sqr() для T = int
double y = sqr(b); // компилятор генерирует sqr() для T = double
```

Наш шаблон `sqr()`, равно как и реализация функции `sqr()` средствами Smalltalk, работает не только со всеми числовыми типами, но и с любыми другими типами, содержащими операцию «*». Тем не менее между этими реализациями есть различие: шаблон `sqr()` подвергается статическому контролю типов, а значит, ошибки при исполнении наподобие «`операция '*' не найдена`» невозможны; версия на Smalltalk, напротив, подвергается динамическому контролю типов, а значит, если `sqr:` будет передан объект, не обеспечивающий операции «*», при исполнении возникнет ошибка. При этом, поскольку в версии на Smalltalk тип `x` в период компиляции не фиксируется, а во время исполнения `sqr:` можно передавать объекты разных типов, этот вариант следует признать более гибким.

Типы как параметры обеспечивают корректный контроль типов в период компиляции. Например, мы можем реализовать процедуру `swap()`, предназначенную для перестановки значений двух переменных, и быть уверены, что их тип будет одним и тем же:

```
template <typename T>
void swap (T &x, T &y)
{ const T z = x; x = y; y = z;
}
//...
int a = 4, b = 7;
swap (a, b); //Все в порядке
double c = 8.0;
swap (a, c); //Ошибка! Поскольку a и c
             // относятся к разным типам,
             // вызов swap неоднозначен
```

Полиморфизм включения или полиморфизм подтипов

Полиморфизм включения (inclusion polymorphism) аналогичен *полиморфизму подтипов* в объектно-ориентированных языках — другими словами, переменные любого данного типа могут хранить объекты его подтипов. Основным инструментом обеспечения полиморфизма подтипов является *переопределение* (overriding) функций, обеспечивающее нескольких реализаций одной и той же виртуальной функции в различных классах. Для пояснения процесса переопределения рассмотрим следующие два класса:

```
class One
{
    public:
        virtual int number()
        {
            return 1;
        }
};
class Two : public One
{
    public:
        virtual int number()
        {
            return 2;
        }
};
```

Класс `One` содержит виртуальную функцию `number()`, возвращающую номер класса — значение 1. Эта функция переопределяется в классе `Two`, который является наследником класса `One`, таким образом, чтобы возвращалось значение 2. Для клиента обе функции представляются семантически как одна полиморфная сущность. Здесь класс `Two` является подтипом класса `One`.

Для полиморфизма подтипов характерна следующая ситуация. В одном из классов имеется общая функция, которая наследуется и используется его потомками. Однако по крайней мере в одном из наследников определена собственная версия функции с тем же именем. Это перекрывает доступ к общей функции для экземпляров данного подкласса (частично или полностью). В случае частичного перекрытия говорят об уточнении функции, в случае полного — о полном ее замещении. В целом же имеют в виду, что вторая функция переопределяет первую.

В качестве универсального инструмента переопределения может рассматриваться отложенная (абстрактная) функция. В такой функции просто нет реализации, она лишь фиксирует идею программиста. Например, для геометрических фигур можно задумать функцию `draw()`, которая их рисует: `Треугольник`, `Окружность` и `Квадрат`. Идею этой функции можно разместить в родительском классе `Фигура`. Однако такая абстрактная операция на самом деле не сможет выполнять какую-то работу, поскольку в классе `Фигура` просто нет данных для любого рисования. Тем не менее наличие функции `draw()` позволяет фиксировать функциональность (рисование) только в классе Фигура, а не вводить три независимые концепции для подклассов `Треугольник`, `Окружность` и `Квадрат`, которые в будущем смогут реализовать эту

идею (путем переопределения). Мало того, подобное решение позволит объявить полиморфную переменную класса `Фигура`, которая будет в различные моменты времени содержать фигуры различного типа. Это вполне допустимо в соответствии с принципом подстановки, который рассматривается в разделе «Природа наследования» главы 16.

Перегрузка и неявное приведение

Под *перегрузкой* (overloading) понимается наличие нескольких реализаций функций, рассчитанных на разные типы операндов при едином имени функции. Так, символ «+» означает сложение и целых, и вещественных чисел. Говорят, что имя функции перегружено, если имеются два (и более) программных тела, связанные с этим именем. При перегрузке полиморфным является имя функции — оно многозначно.

Еще один способ сопоставить перегрузку и полиморфизм: представьте единую абстрактную функцию, которая вызывается с аргументами различного типа, а фактический выполняемый код зависит от типа аргументов. Тот факт, что компилятор часто может определить правильную функцию на этапе компиляции (в языках со строгим контролем типов данных) и, следовательно, сгенерировать только нужный код — это просто оптимизация. Кстати, перегрузка совершенно не подразумевает, что функции, связанные с перегруженным именем, имеют какое-либо семантическое сходство.

Неявное приведение тоже обеспечивает полиморфизм, поскольку автоматически изменяет тип переменной — к примеру, при сложении целого и числа с плавающей точкой целочисленный операнд преобразуется в число с плавающей точкой. Между неявным приведением и перегрузкой могут возникать конфликтные ситуации — в частности, в том случае, если перегруженную функцию можно задействовать с тем же успехом, что и соответствующее приведение.

Чистый полиморфизм имеет место, когда одна и та же функция применяется к аргументам различных типов. В случае чистого полиморфизма есть одна функция (тело кода) и несколько ее интерпретаций. Другая крайность наблюдается, когда имеется множество различных перегруженных функций (то есть тел кода) с одним именем.

Реализация полиморфизма

В языке C++ присутствуют все разновидности полиморфизма [89]. Параметрический полиморфизм выражен в шаблонах, полиморфизм подтипов — в виртуальных функциях, перегрузка — в перегрузке функций, а неявное приведение — во встроенных и пользовательских операциях преобразований. Из них возможность изменчивости в период вычислений обеспечивает лишь полиморфизм подтипов (благодаря динамическому связыванию). Остальные разновидности разрешаются исключительно в период компиляции. В рамках C++ полиморфизм во всех его проявлениях базируется на системе статического контроля типов.

В языке Java ситуация несколько другая. Полиморфизм подтипов и параметрический полиморфизм в этом языке тоже поддерживается. Автоматическое приведение

осуществляется только в отношении встроенных типов — пользовательские типы лишены этой возможности. Перегрузка функций выполняется так же, как и в C++, но при этом программисты лишены возможности перегружать операции. Полиморфизм подтипов представлен в Java в классическом виде. Для представления типов в Java удобно использовать интерфейсы. Интерфейсы аналогичны абстрактным классам C++, содержащим только заголовки чистых виртуальных функций. С помощью интерфейса декларируется тип переменной, которая может указывать на объект любого класса, реализующего этот интерфейс. Интерфейсы играют роль типов данных, оставляющих «за кадром» реализацию операций. Применение интерфейса позволяет сконцентрироваться на задаче реализации операций типа впоследствии. Поскольку в языке C++ такого разделения нет, можно сказать, что C++ поддерживает полиморфизм подклассов (subclass polymorphism), а не подтипов. Впрочем, интерфейсы легко моделируются в C++ с помощью абстрактных классов.

В языках с динамическим контролем типов применение параметрического полиморфизма не имеет смысла. Так как у переменных нет статически объявляемых типов, применение полиморфизма подтипов также оказывается невозможным. При вызове метода объекта Smalltalk в первую очередь проверяет, способен ли данный объект интерпретировать соответствующее сообщение. При положительном результате проверки метод исполняется; в противном случае возвращается сообщение об ошибке. Разновидность полиморфизма, характерная для Smalltalk, не основывается на информации о типах периода компиляции (этого механизма в Smalltalk просто нет) — напротив, разрешение происходит во время исполнения. Этот вид полиморфизма называется *неограниченным*. Перегрузка в языке Smalltalk тоже не имеет смысла, так как нет объявляемых типов переменных. Преобразования типов реализуются с помощью явных методов. Примером может быть метод `asString`.

Более общая форма полиморфизма обеспечивается методами в языках Python и Ruby. Так как переменные в этих языках не имеют типов, у формальных параметров типы тоже отсутствуют. В силу этого метод будет работать с любым типом фактического параметра до тех пор, пока в методе определены операции над формальными параметрами.

Перегруженные подпрограммы

Перегруженная операция имеет несколько интерпретаций. Смысл конкретного экземпляра перегруженной операции определяется типами его операндов. Например, если у операции * языка Java имеются два операнда с плавающей точкой, то она означает умножение чисел с плавающей точкой. Однако если та же операция получает два целочисленных операнда, она означает целочисленное умножение.

Перегруженная подпрограмма — это подпрограмма, имя которой совпадает с именем другой подпрограммы в той же области видимости. Каждая версия перегруженной подпрограммы должна иметь свой уникальный протокол, иначе говоря, она должна отличаться от других версий количеством, порядком, типами своих параметров, а также типом возвращаемого значения (если это функция). Смысл оператора вызова перегруженной подпрограммы задается списком фактических параметров, а также типом возвращаемого значения (при вызове функции).

Языки C++, Java, Ada и C# содержат встроенные перегруженные подпрограммы. Например, многие классы в C++, Java и C# имеют перегруженные операции-конструкторы. Поскольку каждая версия перегруженной подпрограммы имеет уникальный профиль параметров, компилятор может однозначно различать вызовы с различными типами параметров. Конечно, все это совсем не просто. Если разрешены неявные приведения параметров, то процесс распознавания существенно усложняется. А что если после автоматического приведения фактических параметров будет зафиксировано совпадение с профилями параметров нескольких операций? Какая операция должна вызываться? Разработчик языка должен предусмотреть такую возможность и выбрать наиболее вероятный вариант.

Поскольку в языках C++, Java и C# разрешены смешанные выражения, то для однозначного определения вызываемой операции анализируется тип возвращаемого значения. Однако контекст вызова не всегда позволяет определить тип возвращаемого результата. Например, если в C++ две функции с одинаковым именем принимают параметр типа `int`, но одна возвращает результат типа `int`, а другая типа `float`, то компилятор не сможет выявить: какая версия должна быть использована.

В языках Ada, Java, C++, C# и F# программистам также позволяется писать множество версий подпрограмм с одинаковым именем. Более того, в C++, Java и C# большая часть перегруженных операций, создаваемых пользователями, является конструкторами.

К неоднозначным вызовам достаточно часто приводят перегруженные программы с параметрами по умолчанию. К примеру, рассмотрим следующий фрагмент на C++, в котором объявлены две перегруженные процедуры и показан один оператор вызова:

```
void proc(float x = 0.0);
void proc();
. . .
proc();
```

Вызов неоднозначен и приведет к ошибке компилятора.

В языках Ada, C++, Python и Ruby операции могут перегружаться пользователем. Допустим, что в языке Python пишется класс для поддержки арифметических операций над комплексными числами. Комплексное число представляется парой чисел с плавающей точкой. Класс `Complex` должен включать два элемента данных с именами `re` и `im`. Бинарная арифметическая операция здесь задается как вызов метода, направленный первому операнду, причем второй операнд указывается как параметр вызова. Метод сложения имеет имя `__add__`. Например, для вычисления x + y надо записать:

```
x._ _add_ _(y)
```

Для перегрузки сложения, обеспечивающего обработку объектов нового класса `Complex`, нужно создать такое объявление:

```
def _ _add_ _(self, second):
return Complex(self.re + second.re, self.im + second.im)
```

В большинстве языков для объектно-ориентированного программирования ссылка на текущий объект обозначается в вызове неявно. В языке Python эта ссылка указывается явно, поэтому в качестве первого параметра метода `__add__` записано: `self`.

Для языка C++ подобный метод класса `Complex` можно описать в следующей форме:

```
Complex operator + (Complex &second) {
    return Complex(re + second.re, im + second.im);
}
```

Родовые подпрограммы

Повторное использование кода может значительно повысить скорость разработки программных приложений, если есть возможность минимизировать дублирование кода. К примеру, если мы хотим, чтобы подпрограмма сортировки работала с разными типами элементов, нам нужно объявить тип элемента в качестве родового параметра этой подпрограммы. Таким способом мы избавимся от необходимости писать практически идентичный код сортировки для различных типов.

В упомянутой подпрограмме сортировки родовой параметр используется так, что он подходит к различным типам элементов. Следовательно, эта подпрограмма является родовой (generic). Способность кода работать с различными типами именуют полиморфизмом. Как мы уже говорили, один из методов обеспечения полиморфизма предполагает объявление типа элемента в качестве параметра программы. Это называется параметрическим полиморфизмом.

Параметрический полиморфизм поддерживается в период компиляции родовыми (иначе, настраиваемыми или обобщенными) подпрограммами в языках Ada, C++, Java, C# и F#.

Родовые подпрограммы в языке Ada

Основная проблема при многократном использовании готовых подпрограмм состоит в том, что они обрабатывают величины только конкретных типов [26, 52, 62].

Например, следующая процедура обрабатывает величины только типа `Float`:

```
procedure Ord_2 ( A, B : in out Float ) is
    Tmp : Float;
begin
    if A > B then
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Для того чтобы быть действительно полезной программисту, эта процедура должна работать со всеми объектами, к которым применима операция «больше чем». Язык Ada поддерживает определение родовых (настраиваемых) процедур и функций. В них действительные используемые типы определяются пользователем процедуры или функции.

Приведем пример:

```
generic --спецификация
    type T is ( < > ); -- любой дискретный тип
procedure Ord_2 ( A, B : in out T ); -- заголовок Ord_2
procedure Ord_2 ( A, B : in out T ) is --реализация Ord_2
    Tmp : T;
```

продолжение ⇩

```
begin
    if A > B then
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Для передачи параметров настройки в родовую подпрограмму используется механизм формальных родовых параметров. Объявление родовой подпрограммы разделяется на две части: спецификацию, которая определяет интерфейс с внешним миром, и реализацию, которая определяет физическую реализацию.

Формальные родовые типы, которые используются в подпрограмме, указываются в спецификации между словом `generic` и заголовком. В этом примере используется единственный формальный тип `T`. Фактический тип, подставляемый вместо `T`, должен быть одним из дискретных типов языка Ada. На это ограничение указывает обозначение `(< >)` в объявлении `type T is (< >)`. Полный список разновидностей формальных параметров будет приведен чуть позже.

Родовая процедура, по сути, является заготовкой, шаблоном. Перед использованием родовая процедура должна быть конкретизирована, настроена на обработку конкретного типа. Конкретизация задается объявлением:

```
procedure Order is new Ord_2 ( Natural );
```

которое определяет экземпляр-процедуру `Order`, упорядочивающую два параметра типа `Natural`.

Другая родовая процедура может быть написана для упорядочения трех параметров. Внутри ее реализации используется конкретизация процедуры `Ord_2`:

```
generic --спецификация
    type T is ( < > ); -- любой дискретный тип
procedure Ord_3 ( A, B, C : in out T ); -- заголовок Ord_3
with Ord_2;
procedure Ord_3 (A,B,C : in out T) is   --реализация Ord_3
    procedure Order is new Ord_2 ( T ); --конкретизация Order
begin
    Order ( A, B );
    Order ( B, C );
    Order ( A, B );
end Ord_3;
```

Программа, использующая родовую процедуру `Ord_3`, может иметь вид:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ord_3;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
    procedure Order is new Ord_3 ( Natural ); --конкретизация
    Room1 : Natural := 30;  -- квадратных метров
    Room2 : Natural := 25;  -- квадратных метров
    Room3 : Natural := 20;  -- квадратных метров
begin
    Order ( Room1, Room2, Room3 );
    Put ("Комнаты в порядке возрастания площади");
    New_Line;
    Put ( Room1 );  Put ( ' ' );
    Put ( Room2 );  Put ( ' ' );
    Put ( Room3 );
end Main;
```

Существует множество форм формальных родовых параметров, каждая из которых разрешает использование конкретных категорий фактических параметров (табл. 13.2).

Таблица 13.2. Формы формальных родовых параметров

Спецификация формального типа	Фактический параметр может задавать следующий тип	Замечание
is private	Любой нелимитированный тип	1
is limited private	Любой тип	2
is new S	Любой тип, производный от S	1
is new S with private	Тип, производный от тегового типа S	1
is tagged private	Неабстрактный нелимитированный теговый тип	1
is abstract tagged limited private	Любой теговый тип	2
is tagged limited private	Любой неабстрактный теговый тип	2
is interface	Любой интерфейс	1
is interface and A and B	Любой интерфейс и его предки A и B	1
is (< >)	Любой дискретный тип	1
(< >) is private	Неограниченный или неопределенный нелимитированный тип	1, 3
(< >) is limited private	Любой неограниченный или неопределенный тип	2, 3
is mod < >	Любой модульный тип	1
is range < >	Любой целый тип	1
is digits < >	Любой тип с плавающей точкой	1
is delta < >	Любой тип с фиксированной точкой	1
is delta < > digits < >	Любой фиксированный десятичный тип	1
is access S	Ссылочный на S тип	4
is array (Y) of Z	Ограниченный массив	1
is array (Y range <>) of Z	Неограниченный массив	3
with procedure	Процедура с совпадающим профилем	5
with function	Функция с совпадающим профилем	5
with package P is new Q (<>)	Пакет, полученный конкретизацией родового пакета Q	

ЗАМЕЧАНИЯ

1. Использование формального параметра внутри родового модуля ограничено возможностями совместимого с ним фактического параметра.
2. Использование формального параметра ограничено операциями, которые совместимы с лимитированным типом. Таким образом, запрещены такие операции по умолчанию, как присваивание, сравнение на эквивалентность и неэквивалентность.
3. Неопределенным типом называют неограниченный регулярный тип (массив) или неограниченный комбинированный тип (запись), а также защищенный или задачный тип, который не имеет дискриминантов со значениями по умолчанию. Неопределенный тип не может использоваться

для объявления объектов без определения его диапазона (задания значения дискриминанта). Например, объект неопределенного типа:

```
type String is array ( Positive range <> ) of Character;
```

не может быть объявлен без спецификации диапазона.

4. Могут также использоваться следующие ссылочные формы:

```
is access all или is access constant
```

5. Используется для спецификации процедуры или функции, которая применяется в теле родового модуля.

Родовая подпрограмма может настраиваться не только на фактические типы данных, но и на фактические операции. Такая необходимость возникает в двух случаях:

- в подпрограмме намеренно не определена какая-либо обработка (в терминах операций);
- имеющаяся в подпрограмме операция не применима к типу — фактическому параметру настройки.

Рассмотрим второй случай. Например, логическая операция «>», используемая в родовой процедуре `Ord_2`, не определена для типа «запись».

Чтобы расширить область применения, в спецификацию процедуры вносят дополнительный родовой параметр, задающий формальную функцию:

```
generic -- спецификация
    type T is private; -- любой нелимитированный тип
    with function ">" ( A, B : in T )
            return Boolean is <>;  -- требуется определение
    procedure Ord_2 ( A, B : in out T );
procedure Ord_2 ( A, B : in out T ) is   -- реализация
    Tmp : T;
begin
    if A > B then  -- применение операции
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Здесь формальный родовой параметр-функция

```
with function ">" ( A, B : in T )
    return Boolean is <>;  -- спецификация для ">"
```

указывает, что в конкретизации процедуры для экземпляров типа `T` должно быть обеспечено определение операции `">"`. Фраза `"is <>"` в данной спецификации означает, что у этого формального параметра есть значение по умолчанию (определение функции с тем же именем, находящееся в точке конкретизации).

Известно, что к любой программе по умолчанию подключается пакет `Standard`, в котором есть определение функции `">"` для подтипа `Natural`. Поэтому настройка процедуры `Ord_2` на тип `Natural` имеет вид:

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural );
```

то есть второй фактический родовой параметр не указывается, определение функции `">"` по умолчанию берется из пакета `Standard`.

Если же родовая формальная функция записана в форме

```
with function ">" ( A, B : in T )
        return Boolean;  -- спецификация для ">"
```

то формальный параметр не имеет значения по умолчанию, поэтому фактический параметр-функция с сигнатурой `function (A, B : in T)` должна быть определена при конкретизации:

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural, ">");
```

Наконец, мы можем перекрыть функцию `">"`. Для этого зададим конкретизацию в виде

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural, "<" );
```

В этом случае экземпляр `Order` будет размещать элементы в убывающем порядке.

В заключение приведем пример программы, использующей настройку родовой процедуры `Ord_2` на тип «запись».

```
with Ada.Text_IO, Ord_2;
use Ada.Text_IO;
procedure Main is
    type Age_Type is range 0 .. 120;
    type Person is record
        Name : String (1..10) := (others => ' ');
        Age : Age_Type;
    end record;
    function ">" ( C, D : in Person ) return Boolean is
        begin
            return C.Age > D.Age;
        end ">";
    procedure Order is new Ord_2 ( Person ); -- конкретизация
    Person1 : Person := ( "Александр", 50 );
    Person2 : Person := ( "Николай", 25 );
begin
    Order ( Person1, Person2 );
    Put ("Список сотрудников в
                порядке возрастания возраста: ");
    Put ( Person1.Name ); Put (' ');
    Put ( Person2.Name ); New_Line;
end Main;
```

Родовые подпрограммы в языке C++

Родовые функции в языке C++ называются функциями-шаблонами. Объявления функции-шаблона имеет следующую общую форму:

```
template <параметры шаблона>
– объявление функции, которая может иметь параметры шаблона
```

Запись в угловых скобках аналогична списку параметров в обычной функции.

Функция-шаблон должна иметь хотя бы один родовой параметр, представляемый в одной из следующих форм:

```
1.    class имя_родового_параметра
2.    typename имя_родового_параметра
```

Первую форму целесообразно использовать для задания параметров, которые являются именами классов. Вторая форма применяется для указания имени типа. Первая форма была придумана давно, а вторая форма появилась много позже и более прозрачно отражает ее смысл, поскольку родовой параметр обозначает место для подстановки имени фактического типа. В данном контексте стандарт C++ рассматривает оба эти ключевых слова как идентичные.

В качестве примера повторим объявление функции-шаблона, приведенной в подразделе «Параметрический полиморфизм»:

```
template <typename T>
void swap (T &x, T &y)
{ const T z = x; x = y; y = z;
}
```

Эта функция-шаблон содержит один формальный родовой параметр `T` (указывает тип данных, с которыми работает функция) и два обычных формальных параметра-ссылки `x` и `y`. Экземпляры данной функции-шаблона можно создавать для работы с любым типом данных, для которого определена операция присвоения. Например, экземпляр подпрограммы для типа `int` в качестве параметра может иметь следующий вид:

```
void swap (int &x, int &y)
{ const int z = x; x = y; y = z;
}
```

Еще один пример:

```
template <typename T>
T max(T a, T b)
    return a > b ? a : b;
}
```

Здесь тоже один формальный родовой параметр `T` и два обычных формальных параметра `a` и `b`. По параметру `T` функция настраивается на тип обрабатываемых данных, а по параметрам `a` и `b` — на их значения. В теле функции использован условный оператор `?`, логика работы которого состоит в анализе условия, стоящего слева от знака вопроса, и (при его истинном значении) возврате (в качестве результата) значения первого операнда, стоящего справа от знака вопроса. В противном случае возвращается значение второго операнда, стоящего после двоеточия. По сути, оператор `?` моделирует логику работы обычного оператора `if-then-else`, но имеет «загадочный» внешний вид и лаконичную форму. Он удобен в тех случаях, когда в тексте программы надо указать единственное место появления «условного» результата.

Экземпляры функции-шаблона в языке C++ создаются неявно, при обработке оператора вызова функции. Причем абсолютно неважно, что именно указано в операторе вызова функции: ее имя или ее адрес, полученный с помощью операции `&`. Например, экземпляр функции-шаблона `max` в приведенном фрагменте создается дважды — для параметров типа `int` и типа `char` соответственно:

```
int a, b, c;
char x, y, z;
c = max(a, b);
z = max(x, y);
```

Объявим процедуру-шаблон для сортировки массивов любого типа:

```
template <typename T>
void generic_sort(T arr[], int size {
    int top, bottom;
    T temp;
    for (top = 0; top < size - 2; top++)
        for (bottom = top + 1; bottom < size - 1; bottom)
            if (arr[top] > arr[bottom]) {
                temp = arr[top];
                arr[top] = arr[bottom];
                arr[bottom] = temp;
            }//* конец цикла for ( bottom
}//** конец родовой подпрограммы сортировки
```

Экземпляр этого шаблона появляется, например, при обработке следующего вызова:

```
float arr_float[500];
generic_sort(arr_float, 500);
```

Подпрограммы-шаблоны в языке C++ отдаленно напоминают подпрограммы, в которых типы формальных параметров при вызове динамически связываются с типами фактических параметров. В «динамических» подпрограммах нужен лишь один экземпляр кода, в то время как шаблоны требуют создания (в период компиляции) копии кода для каждого отдельного типа, кроме того, связывание операторов вызова с подпрограммами также осуществляется статически.

Родовые методы в языке Java

Поддержка родовых типов и методов была добавлена в язык Java на уровне пятой версии. Заголовок родового класса включает имя, за которым в угловых скобках записываются переменные типов:

```
generic_class<T>
```

здесь `T` — это переменная типа.

Обсудим отличия родовых методов в Java от родовых подпрограмм в C++:

1. Родовые параметры должны быть классами, они не могут быть элементарными типами. Следовательно, в Java нельзя точно смоделировать родовую сортировку массивов, рассмотренную нами применительно к языку C++, где элементы массива имели родовой тип, а могли быть и элементарного типа.
2. Родовые методы могут конкретизироваться многократно, но все равно создается только одна копия кода. Внутренняя версия родового метода, называемая сырым (raw) методом, работает с объектами самого общего класса — `Object`. Лишь для того момента, когда из родового метода должен возвращаться результат, компилятор вставляет код явного приведения значения к правильному типу.
3. Для родового метода могут быть определены ограничения (границы): диапазон классов, которые могут использоваться в родовых параметрах.

Обсудим следующий родовой метод на Java:

```
public static <T> T worker(T[] list) {
. . .
}
```

В методе `worker()` задан один обычный параметр `list` и один родовой параметр — тип `T`, который должен быть массивом. Конкретизация этого метод вызывается следующим оператором:

```
worker <String>(theList);
```

Ограничения на родовой параметр метода записываются следующим образом:

```
public static <T extends Comparable> T worker(T[] list) {
. . .
}
```

Здесь определен метод, чьим родовым параметром является массив из элементов класса, который реализует интерфейс `Comparable`. Обычно ключевое слово `extends` означает, что у родового класса есть подкласс, имя которого записано за ключевым словом. В данном контексте `extends` имеет другой смысл. Выражение `<T extends Ограничивающий_тип>` означает, что `T` должен быть подтипом для типа `Ограничивающий_тип`. Следовательно, здесь `extends` означает, что родовой класс (интерфейс) или расширяет ограничивающий класс (если границы заданы классом), или реализует ограничивающий интерфейс (если границы заданы интерфейсом). Ограничение гарантирует, что элементы любого экземпляра родового метода можно будет сравнивать с помощью метода `compareTo()` из интерфейса `Comparable`.

При наличии нескольких ограничений на родовой тип родового метода они перечисляются после предложения `extends`, отделяясь одно от другого знаком амперсанда (&). Отметим, что родовой метод может иметь более одного родового параметра.

Язык Java поддерживает *подстановочные* (wildcard) типы. Например, `Collection<?>` является подстановочным типом для коллекции классов `Collection`. Этот тип может быть использован для любых компонентов класса в любом типе коллекции. К примеру, рассмотрим следующий родовой метод:

```
void printCollection(Collection<?> c) {
    for (Object e: c) {
        System.out.println(e);
    }
}
```

ПРИМЕЧАНИЕ

Подстановочный символ «знак вопроса» разрешает одну подстановку в своей позиции. Когда говорят, что переменная x имеет тип List<?>, это значит, что существует некоторый тип T, для которого x имеет тип List<T>.

Этот метод печатает элементы из любого класса коллекции, невзирая на то, к какому классу они принадлежат. При работе с объектами подстановочного типа нужно соблюдать некоторую осторожность. Например, поскольку компоненты конкретного объекта данного типа тоже имеют тип, объекты любого другого типа не смогут быть добавлены в коллекцию. Рассмотрим оператор:

```
Collection<?> c = new ArrayList<String>();
```

Он запрещает использовать метод `add` для того, чтобы положить что-нибудь в эту коллекцию, если добавление не имеет типа `String`.

Подстановочные типы могут ограничиваться ровно так же, как и неподстановочные типы. Такие типы называют ограниченными подстановочными типами. Обратимся к следующему заголовку метода:

```
public void drawAll(ArrayList<? extends Shape> things)
```

Родовым типом здесь считается подстановочный тип, который является подклассом класса `Shape`. Этот метод может быть написан для рисования любого объекта, чей тип является подклассом класса `Shape`.

Родовые методы в языке C#

Родовые методы в C# (версии 2005 года) схожи по своим возможностям с аналогичными методами в Java, за исключением того, что не поддерживают подстановочные типы [38]. Здесь обеспечивается одна уникальная возможность: в операторе вызова родового метода типы фактических параметров могут опускаться, если компилятор способен самостоятельно «вывести» опущенный тип. Рассмотрим следующий фрагмент:

```
class theClass {
    public static T worker<T>(T param) {
    . . .
    }
}
```

Метод `worker()` может быть вызван без определения родового параметра, если компилятор может его вывести из фактического параметра в операторе вызова. К примеру, следующие вызовы считаются легальными:

```
int theInt = theClass.worker(47);         // Вызов  worker<int>
string theStr = theClass.worker('world'); // Вызов worker <string>
```

Родовые функции в языке F#

Система логического вывода типа в языке F# иногда не способна определить тип параметров и тип возвращаемого из функции результата. В подобных случаях F# выводит родовой тип параметров и результата. Это называется автоматическим обобщением. Рассмотрим следующее определение функции:

```
let readLast (x, y, z) = z;;
```

Поскольку информации о типах здесь нет, выводятся родовые типы параметров и результата. Эта функция не содержит каких-то вычислений, она является простой родовой функцией. Для функций допускается наличие родовых параметров. Приведем пример:

```
let printPair (x: 'a) (y: 'a) = printfn "%A %A" x y;;
```

Здесь `%A` — это спецификация формата для любого типа. Апостроф перед именем задает для него родовой тип. Данное определение функции работает с родовыми параметрами, так как отсутствуют операции с ограничениями типов.

Арифметические операции являются примерами операций с ограничениями по типам. К примеру, рассмотрим следующее определение функции:

```
let adder x y = x + y;;
```

Логический вывод типа установит для `x`, `y` и результата тип `int`. Поскольку в языке F# отсутствует неявное приведение типа, следующий вызов считается корректным:

```
adder 2.5 3.6;;
```

Даже при задании родовых типов параметров операция + приведет типы `x` и `y` к формату `int`.

Родовой тип может быть определен явно, с помощью угловых скобок:

```
let printPair2<'T> x y = printfn "%A %A" x y;;
```

В вызове этой функции должен явно задаваться тип:

```
printPair2<float> 3.5 2.4;;
```

В силу наличия логического вывода типов и отсутствия неявного приведения, родовые функции F# значительно менее полезны для числовых вычислений, чем аналогичные функции в C++, Java и C#.

Замыкания

Определить замыкание проще простого: *замыкание* — это подпрограмма и область видимости, где она объявлена [36]. Подпрограмма может быть вызвана из любой точки этой области. Объяснить суть замыкания совсем не просто.

В языках со статической областью видимости, где запрещены вложенные подпрограммы, замыкания не нужны, такие языки их не поддерживают. Все переменные в области видимости подпрограммы такого языка доступны, независимо от места в программе, откуда вызывается подпрограмма. Это справедливо и для локальных переменных подпрограммы, и для глобальных переменных.

Когда подпрограммы могут быть вложены, в добавление к локальным и глобальным переменным область видимости может включать переменные, которые определены во всех объемлющих подпрограммах. Здесь нет ничего особенного, если подпрограмма может вызываться только из тех точек, где объемлющие области видимости активны и видимы. Проблема проявляется, если подпрограмма может быть вызвана еще откуда-то. Например, если подпрограмма может передаваться как параметр или присваиваться переменной, то это значит, что она может быть вызвана из любой точки программы. При этом возникает еще одна проблема. Подпрограмма может быть вызвана после завершения некоторых из ее вложенных подпрограмм, это обычно означает, что переменные из этих подпрограмм уничтожены — они просто не существуют. Если подпрограмма может быть вызвана из любой точки программы, то ее область видимости должна быть видима (доступна) из любой точки. Следовательно, время жизни переменных, объявленных во вложенных программах, не должно ограничиваться периодом активности этих подпрограмм, а должно соизмеряться со временем жизни программы. Переменные, время жизни которых определяется временем жизни программы, называют *неограниченными*. Как правило, они должны размещаться в куче, а не в стеке.

Почти все функциональные языки, большинство скриптовых языков и один объектно-ориентированный язык (C#) поддерживают замыкания. Эти языки характеризуются статической областью видимости, разрешают вложенные подпро-

граммы и передачу подпрограмм как параметров. Следующий пример замыкания написан на JavaScript:

```
function counter(x) {
    return func(y) {return x + y;}
}
. . .
    var add47 = counter(47);
    var add82 = counter(82);
    document.write("Add 47 to 13: " + add47(13) + "<br />");
    document.write("Add 82 to 13: " + add82(13) + "<br />");
```

Будем считать, этот фрагмент встроен в документ HTML, а результаты вычислений отображаются браузером в следующей форме:

```
Add 47 to 13: 50
Add 82 to 13: 95
```

В этом примере замыканием является анонимная функция `func`, которая определена внутри функции `counter` и обеспечивает возврат значения из `counter`. Переменная `x`, на которую ссылаются в замыкающей функции, связана с параметром, который передается функции `counter`.

Функция `counter` вызывается дважды, один раз с параметром `47`, а другой раз — с параметром `82`. Каждый из этих вызовов возвращает собственную версию замыкания, поскольку связан со своим значением `x`. Первый вызов `counter` создает функцию, которая добавляет `47` к ее параметру; второй вызов создает функцию, которая добавляет `82` к ее параметру. Обе версии функции связаны с различными активациями функции `counter`. Очевидно, что время жизни версии `x`, создаваемой при вызове `counter`, должно быть расширено на все время жизни программы.

Такая же замыкающая функция может быть написана на C# с помощью вложенного анонимного делегата. Тип вложенного метода определяется как функция, принимающая параметр типа `int` и возвращающая анонимный делегат. Тип результата определяется специальной нотацией для таких делегатов, `Func<int, int>`. Первым в угловых скобках записан тип параметра. Такой делегат может инкапсулировать методы только с одним параметром. Вторым в угловых скобках записан тип результата, возвращаемого из метода, инкапсулированного делегатом.

```
static Func<int, int> counter(int x) {
    return delegate(int y) { return x + y;};
}
. . .
Func<int, int> Add47 = counter(47);
Func<int, int> Add82 = counter(82);
Console.WriteLine("Add 47 to 13: {0}", Add47(13));
Console.WriteLine("Add 82 to 13: {0}", Add82(13));
```

Результаты вычислений у этого фрагмента те же самые, что и в примере на JavaScript.

Анонимный делегат может быть записан в виде лямбда-выражения. Вместо делегата в тело метода `counter` можно поместить следующее лямбда-выражение:

```
return y => x + y
```

Блоки в языке Ruby реализованы так, что могут ссылаться на переменные, видимые в точке определения блока, даже если они вызываются в точке, где эти переменные должны отсутствовать. Подобные блоки являются замыканиями.

Контрольные вопросы и упражнения

1. Чем процедура отличается от функции?
2. Чем процедура схожа с функцией?
3. Из каких элементов состоит описание процедуры?
4. Каковы особенности выполнения рекурсивной процедуры?
5. В чем состоят преимущества использования процедур?
6. Что такое формальный параметр?
7. Что такое фактический параметр?
8. Что такое метод передачи параметров?
9. Чем отличаются различные методы передачи параметров?
10. В чем суть метода передачи по значению?
11. Какие достоинства и недостатки имеет передача по значению?
12. В чем суть метода передачи по ссылке?
13. Дана Pascal-процедура, суммирующая элементы двух целых массивов. Используются два формальных параметра — имена массивов. Результат суммирования заносится во второй массив. Способ передачи параметров — вызов по ссылке.

```
Type vector = array[1..5] of integer;
Var arr1, arr2 : vector;
Procedure Adder (var a : vector; var b : vector);
Var i : integer;
begin
    for i := 1 to 5 do
        b[i] : = a[i] + b[i]
end;
```

    Переписать эту процедуру на языке C.
14. Дана процедура, в которой передача параметров выполняется по имени:

```
Procedure BigSub;
Var global : integer; list : array [1..2] of integer;
Procedure SMallSub (var local : integer); {формальные параметры}
begin
    local := 3;
    global := global + 1;
    local := 5;
end; {SmallSub}
begin {BigSub}
    list [1] := 3;
    list [2] := 1;
    global := 1;
    SmallSub (list [global]) {фактические параметры}
end; {BigSub}
```

    Записать шаги вычислений и указать значения в массиве `list`, формируемые в `BigSub` после возврата из `SmallSub` для следующих методов передачи параметров:

    а) передача по значению;

    б) передача по ссылке;

в) передача по значению-результату (copy-in/copy-out);
г) передача по имени.

Результаты поместить в следующую таблицу.

Метод	list[1]	list[2]
Передача по значению		
Передача по ссылке		
Передача по значению-результату		
Передача по имени		

15. Дана следующая С-программа:

```
void  main  (  )
{
    int value = 1, list [2] = {3,5};
    swap (value, list [0]);
    swap (list [0], list [1]);
}
void swap (int a; int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Записать шаги вычислений и указать, какими будут значения переменных `value`, `list[0]` и `list[1]` после каждого из двух вызовов `swap` при условии, что параметры передаются:
а) по значению;
б) по ссылке;
в) по значению-результату (copy-in/copy-out);
г) по имени.

Результаты поместить в следующую таблицу.

Метод	Вызов 1			Вызов 2		
	value	list[0]	list[1]	value	list[0]	list[1]
Передача по значению						
Передача по ссылке						
Передача по значению-результату						
Передача по имени						

Глава 14
Управление подпрограммами

В этой главе определяются общие понятия и механизмы взаимодействия с подпрограммами, затрагивающие их размещение в памяти и связывания, которые происходят при выполнении подпрограмм. Читатель знакомится с эффектом вложения областей видимости, который возникает при помещении подпрограммы в контекст программы, после этого его внимание концентрируется на активации подпрограммы в ходе ее вызова. Поясняется поток управления между активациями подпрограмм, иллюстрируемый деревом активации, а также формат записи активации. Описываются особенности размещения записи активации в куче и в стеке, а также освобождение кучи (стека) после завершения выполнения подпрограммы. Далее приводятся примеры управления подпрограммами в языках C и Pascal, а также объясняется методика оптимизации взаимодействия с подпрограммой на основе дисплеев.

Вложенные области видимости объявлений

Будем различать связывающее и ограниченное вхождения имени.

Связывающее вхождение имени вводит новый смысл его использования. Остальные вхождения этого имени называют *ограниченными*.

Связывающее вхождение является *объявлением* имени.

Правила области видимости определяют: какое объявление применимо к ограниченным *вхождениям* имени. Говорят, что каждое ограниченное вхождение имени находится в области видимости его связывающего вхождения (объявления) [31, 32, 34, 38, 42, 76].

Связи от ограниченных вхождений к объявлениям, в области видимости которых они находятся, покажем пунктирными стрелками (рис. 14.1).

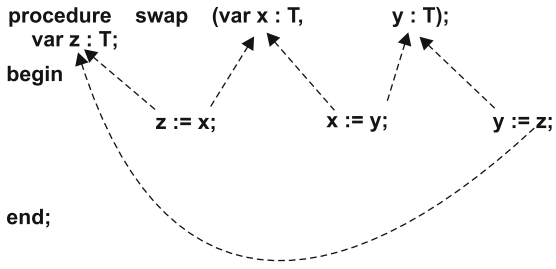


Рис. 14.1. Связи области видимости

Это объявление процедуры содержит объявления ее имени swap, имен формальных параметров x, y, а также имени переменной z в теле. Области видимости x, y и z включают тело процедуры swap. Вне процедуры может быть использовано имя процедуры. В теле процедуры присутствуют по паре ограниченных вхождений имен x, y и z.

Приведенный рисунок иллюстрирует картину связываний с точки зрения ограниченных вхождений. Это, так сказать, точка зрения компилятора, решающего вопрос: какой смысл имеет встреченное им имя. Если же рассматривать точку зрения объявлений (какие вхождения они видят в своих областях видимости), то направления всех стрелок следует переменить на противоположное.

Вхождения имен swap, x и y в строке

```
procedure swap (var x, y : T)
```

являются связывающими вхождениями.

Пары вхождений x, y и z

```
procedure swap (x, y : T);
var  z : T;
begin
    z := x;  x := y;  y := z
end;
```

между операторными скобками begin и end в теле процедуры считаются ограниченными вхождениями.

В терминах связывающего и ограниченного вхождений можно объяснить использование терминов «объявление», «имя». Объявление x является связывающим вхождением x. На практике об объявлении x говорят как об имени x.

Таким образом, утверждение

```
"область видимости формального параметра x содержится внутри процедуры swap"
```

имеет точный аналог

```
"область видимости объявления x как формального параметра содержится внутри
процедуры swap"
```

Объявление имени n видимо в данной точке программы, если вхождение n в этой точке может быть в области видимости этого объявления. Когда конкретное объявление № ясно из контекста, мы говорим «имя n видимо» (имея в виду «объявление № имени n видимо».

Имя является *локальным* или *ограниченным* внутри фрагмента программы, если оно видимо в этом фрагменте.

В языке C говорят, что нелокальное имя является *глобальным*, так как оно видимо во всех подпрограммах.

Вложения конструкций, в которых разрешено объявление имен, образуют вложенные области видимости.

Например, в языке C разрешены объявления внутри составных операторов.

Составные операторы группируются в скобках { и }, необязательные объявления могут появляться после открывающей скобки:

```
{<список объявлений> <список операторов>}
```

Такую группировку объявлений и операторов будем называть *блоком*. В C не разрешено объявлять одни подпрограммы внутри других, поэтому объявления в блоке не могут содержать объявления подпрограмм.

Объявления внутри составных операторов позволяют объявлять имена везде, где это необходимо. Приведем пример блока, где переменная *i* объявлена рядом с тем местом, где она используется для управления числом повторений тела цикла:

```
{
    int i;
    i = 0;
    while (i <= max) {
        i = i + 1;
    }
}
```

Сами составные операторы могут вкладываться друг в друга (рис. 14.2).

На рис. 14.2 для выделения вложенности составные операторы заключены в прямоугольники. Вложенность операторов показывается отступами (большей вложенности соответствует больший отступ).

Так же как операторы вкладываются друг в друга, так и области видимости вкладываются друг в друга.

Областью видимости объявления x в операторе является этот оператор, включая любые вложенные операторы, в которых x не переопределено.

Переопределение `x` создает дырку в области видимости любых объемлющих объявлений `x`.

На приведенном ниже рисунке цифры во вхождениях имени *i* не являются частью программного текста, они просто позволяют нам говорить о трех вхождениях. Объявления `i2` и `i3` создают дырки в области видимости `i1`. Эти дырки выделены темным цветом. В области видимости объявления `c` дырок нет, так как `c` не переопределено.

В языке Pascal роль блоков играют подпрограммы. Разрешено их вкладывать друг в друга:

```
program Main;
  - Локальные объявления для Main;              Начало Main
  procedure Sub1;                                          |
    - Локальные объявления для Sub1;        Начало Sub1    |
    procedure Sub3;                                  |     |
       - Локальные объявления для Sub3; Начало Sub3  |     |
       begin                                     |   |     |
       - Операторы для Sub3;                     |   |     |
       end {Sub3};                         Конец Sub3  |     |
    procedure Sub4;                                  |     |
      - Локальные объявления для Sub4;  Начало Sub4  |     |
      begin                                     |    |     |
      - Операторы для Sub4;                     |    |     |
      end {Sub4};                         Конец Sub4 |     |
    begin                                            |     |
    - Операторы для Sub1                             |     |
    end {Sub1};                               Конец Sub1    |
  procedure Sub2;                                          |
    - Локальные объявления для Sub2         Начало Sub2    |
    begin                                            |     |
    - Операторы для Sub2                             |     |
    end {Sub2};                               Конец Sub2    |
begin                                                      |
  - Операторы для Main;                                    |
end {Main }.                                  Конец Main
```

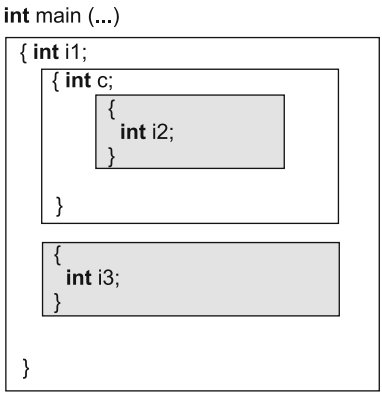


Рис. 14.2. Вложения составных операторов и областей видимости

Здесь вложенные области видимости образуются при вложении друг в друга объявлений подпрограмм: ведь в них могут содержаться объявления имен (рис. 14.3).

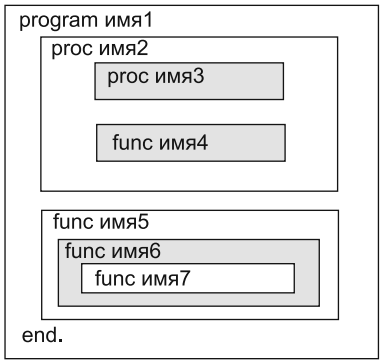


Рис. 14.3. Вложения подпрограмм и областей видимости

В подпрограмме область видимости объявления продолжается от точки объявления до конца подпрограммы.

Например, в программе с процедурами

```
Program U;
    Procedure X;
    Procedure Y;
    …
    end Y;
    end X;
    Procedure A;
    Procedure B;
    …
    end B;
end A;
…
```

объявление переменной в программе U видимо во всех процедурах X, Y, A, B и в самой программе U. Объявление в X видимо в X и Y. Объявление в Y видимо только в Y.

Эти правила видимости применимы и к процедурным именам. Процедурное имя X видимо во всех процедурах X, Y, A и B, но имя процедуры Y не видимо из A, B.

На все блочно-структурированные программы распространяются следующие *правила статической области видимости*:

1. Объявления в заголовке блока локальны в его пределах. Любое вхождение имени в пределах тела блока (не включая вложенные подблоки) рассматривается как ссылка на локальное объявление имени.
2. Если в пределах тела блока имеется вхождение имени, но локальное объявление имени отсутствует, то это вхождение рассматривается как ссылка на объявление в объемлющем блоке, который включает в себя данный блок. Если и в нем искомое объявление имени отсутствует, то поиск продолжается в блоке следующего уровня и т. д. Если же объявление не будет найдено и в самом внешнем (верхнем) блоке, то имя либо относится к предопределенным именам данного языка и его объявление берется из среды предопределенных имен, либо (если такого имени нет и среди предопределенных имен) эта ситуация рассматривается как ошибка. Следовательно, среда предопределенных имен языка считается блоком, внешним по отношению к самому внешнему блоку программы.
3. Если блок содержит объявление другого блока, то все локальные объявления в этом внутреннем блоке или блоках, содержащихся внутри него, полностью скрыты от внешнего блока, в котором не могут появиться ссылки на эти объявления. Таким образом, внутренние блоки инкапсулируют свои локальные объявления, делая их невидимыми для внешних блоков.
4. Блок может иметь имя (обычно если он представляет собой именованную подпрограмму). Имя блока входит в число локальных имен объемлющего блока. Например, если в главной программе на языке Pascal объявляется процедура с заголовком

   ```
   procedure P (A : real);
   ```

   то имя процедуры P является локальным именем в главной программе, тогда как имя формального параметра A является локальным именем в самой процедуре P. В пределах главной программы можно ссылаться на P, но не на A.

По этим правилам можно несколько раз объявлять одно и то же имя в различных блоках, но объявление во внешнем блоке всегда становится скрытым внутри вложенного блока, если в последнем это имя объявляется заново.

Правила статической области видимости для блочно-структурированных программ позволяют связывать каждое вхождение имени в пределах одного блока с определенным объявлением этого имени в процессе компиляции подпрограммы. Программисту требуется лишь создавать правильные объявления (в пределах каждого блока), а также правильно вкладывать блоки друг в друга. На основе правил статического контроля компилятор языка всегда способен провести статическую проверку. Все это послужило причиной широкой распространенности блочной структуры программы во многих языках программирования.

Связывания при выполнении подпрограмм

В процессе выполнения подпрограмм происходит ряд связываний (рис. 14.4):

1. Имя оператора вызова подпрограммы связывается с ее объявлением. Это происходит в период компиляции.
2. Объявления переменных в описании подпрограммы связываются с конкретными областями в памяти, где они размещаются. Так выполняется активация подпрограммы, она происходит в период активации.

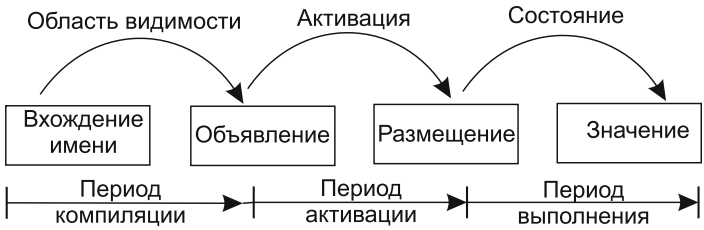


Рис. 14.4. Связывания при выполнении подпрограммы

3. В период выполнения подпрограммы происходит связывание областей ее размещения с вычисляемыми значениями.

Поток управления между активациями подпрограмм

При вызове подпрограммы происходит ее активация. При этом в памяти формируется структура — запись активации. Запись активации содержит все данные, необходимые для работы подпрограммы. В каждый момент времени управление может передаваться только одной подпрограмме. При вызове из `P` подпрограммы `Q` выполнение `P` приостанавливается. Выполнение `P` возобновляется при возврате управления из `Q`. Таким образом, передача управления между активациями подпрограмм соответствует принципу LIFO (последний вошел — первый вышел).

Поток управления между активациями можно проиллюстрировать вставкой в тела подпрограмм операторов записи.

В следующей функции факториала управление достигает оператора записи сразу после входа и перед выходом из тела функции:

```
function  f (n : integer) : integer;
    begin
    writeln ('→ f(', n, ')');
    if   n = 0   then   f := 1  else  f := n * f (n - 1);
    writeln  (' ← f(', n, ')')
    end;
```

Здесь операторы записи трассируют поток управления между активациями. Символ → можно читать как «начало активации», а символ ← как «завершение активации».

Покажем трассировку для вызова `f(3)`. Для иллюстрации вложенности вызовов используем отступы (в функции средства организации отступов не показаны, но их легко добавить):

```
→ f (3)
    → f (2)
    → f (1)
        → f (0)
        ← f (0) = 1
    ← f (1) = 1
    ← f (2) = 2
← f (3) = 6
```

Видим, что выполнение активации `f(3)` приостанавливается, пока управление находится в активациях `f(2)`, `f(1)` и `f(0)`.

Деревья активации

Поток управления между активациями подпрограмм может быть отображен с помощью дерева активации. Вершины в дереве представляют активации. Если активация `P` вызывает активацию `Q`, то у вершины `P` есть потомок — вершина `Q`.

Если `P` вызывает `Q` перед тем как вызвать `R`, вершина для `Q` появляется левее вершины для `R`.

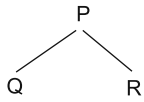


Рис. 14.5. Дерево активации

Пример 14.1. Рассмотрим вариант функции факториала, которая вычисляет `n – 1` вызовом функции `pred(n)`:

```
program  Activations;
    function pred (m : integer) : integer;
    begin
        pred := m – 1
    end;
    function f (n : integer) : integer;
    begin
        if  n = 0  then  f := 1  else  f := n * f (pred (n))
    end;
Begin
    f (3)
End.
```

Активации, происходящие при выполнении этой программы, проиллюстрируем деревом активаций (рис. 14.6). Фактические параметры вызова функции будем записывать внутри скобок. В этом примере параметры передаются по значению. Идея дерева активации независима от метода передачи параметров.

Выполнение программы начинается с активации `f(3)`. Так как условие `3 = 0` ложно, управление передается оператору присваивания с правой частью

```
n * f ( pred (n) )
```

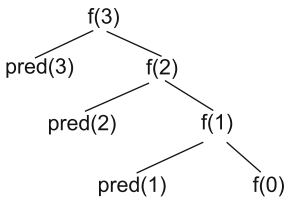


Рис. 14.6. Дерево активаций программы Activations

При `n = 3` это выражение вычисляется вызовом `pred(3)`. После возврата управления из `pred(3)` с результатом `2` происходит вызов `f(2)`. Аналогичным образом вызов `f(2)` приводит к вызову `pred(2)`, далее вызывается `f(1)` и т. д.

Формат записи активации

Запись активации — это шаблон (фрейм), который содержит все данные, необходимые для активации подпрограмм. Эта запись содержит память для локальных переменных, формальных параметров, а также для любой дополнительной информации, требуемой для активации [1, 32, 49, 76, 81].

Обобщенный формат записи активации представлен на рис. 14.7.

Рис. 14.7. Обобщенный формат записи активации

В каждом языке принята своя детализация структуры записи активации.

Рассмотрим пример:

```
function abc ( u : Link ) : Boolean;
    var    i : integer;  c : char;
    begin
        …
    end;
```

В этом случае запись активации должна содержать память для хранения формального параметра `u`, булева результата, локальных переменных `i` и `c`, временную память, требуемую для вычисления выражений в теле функции, а также память для управления активациями.

Локальные данные записи активации

Каждая активация функций `f( )` и `pred( )` из примера 14.1 требует памяти для формальных параметров. Формальный параметр `n` локален для `f( )`. Каждой рекурсивной активации `f(3)`, `f(2)`, `f(1)` требуется собственная память для формального параметра `n`. Поскольку активация `f(2)` существует во время активации `f(3)`, память для формального параметра `n` в `f(2)` должна быть отлична от памяти для `n` в `f(3)`.

Таким образом, каждая активация рекурсивной подпрограммы нуждается в собственной памяти для ее формальных параметров. Аналогично каждой активации требуется отдельная память под локальные переменные.

Связи управления и доступа

Различают два вида связей между записями активации:

- *Связи управления* (динамические связи) — указывают на запись активации вызывающей подпрограммы (периода выполнения).
- *Связи доступа* (статические связи) — обеспечивают доступ к внешним переменным, их объявлениям. Применяют при реализации языков со статической областью видимости.

Управляющая связь указывает на вызывающую подпрограмму. Если `P` вызывает `Q`, то управляющая связь в активации `Q` будет указывать на активацию вызывающей `P`.

Пример 14.2. Проиллюстрируем необходимость связей доступа.

Повторим пример программы, которая обеспечивает различный вывод (по правилам статической и динамической области видимости).

```
program L;
    var n : char;          {n объявлена в L}
    procedure W;
        begin
            writeln (n)    {вхождение n в W}
        end;
    procedure D;
        var n : char;      {n переобъявлена в D}
        begin
            n := 'D';
            W              {W вызывается внутри D}
        end;
Begin                      {начало L}
    n := 'L';
    W;                     {W вызывается из главной программы L}
    D                      {D вызывается из главной программы L}
End.
```

Согласно статической области видимости, нелокальная переменная `n` в процедуре `W( )` находится в области видимости самого внешнего объявления `n` в программе. Согласно динамической области, первый вызов `W( )`, выполняемый из `L`, связывает `n` с самым внешним объявлением, а второй вызов `W( )` (из `D`) связывает `n` с объявлением в `D( )`.

Вычисления, выполняемые программой, иллюстрирует дерево активаций на рис. 14.8.

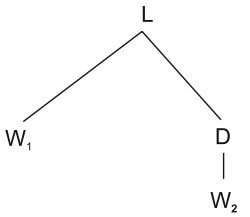


Рис. 14.8. Дерево активаций программы L

С помощью индексов будем различать активации процедуры W (W_1 и W_2). Главная программа прямо вызывает W_1. Далее она вызывает D, которая вызывает W_2.

Покажем записи активации, каждая из которых соответствует определенной точке вычислений: в языках Pascal и C память из-под записи активации будет освобождена перед созданием записи активации для D (рис. 14.9).

Стрелки между записями активации соответствуют управляющим связям.

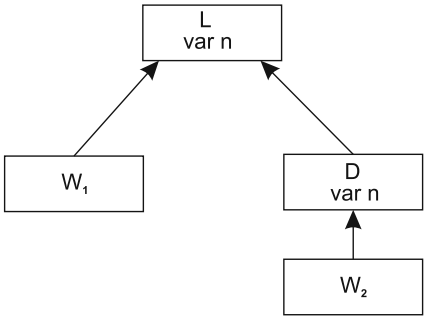


Рис. 14.9. Записи активаций для программы L

Так как n является для W нелокальной переменной, память для n должна находиться в другой записи активации. Согласно динамической области видимости, память для n может быть найдена по управляющей связи, указывающей на ближайшее связывание n. Для записи активации W_1 ближайшее связывание n находится в записи активации L, а для записи активации W_2 ближайшее связывание n находится в записи D.

О статической области видимости. В языке Pascal используется явная связь доступа. Связь доступа указывает из записи активации W на L, так как в исходном тексте процедура W появляется внутри L.

Размещение и освобождение в куче

Память под записи активации может выделяться в области, называемой кучей (heap). Куча — особая область памяти, у которой имеются свои правила управления. Записи остаются в куче, пока они нужны. Для автоматического освобождения этой памяти выполняется «сборка мусора».

Время жизни записи активации в куче начинается с ее размещения и заканчивается тогда, когда исчезает необходимость в ее использовании.

При размещении в куче время жизни записи активации не нужно связывать с потоком управления LIFO между активациями.

Даже после возврата управления из подпрограммы P ее запись активации может оставаться в куче сколь угодно долго.

В языках Algol 60, Pascal, C традиционно используется размещение в стеке. Использование кучи для размещения записей активации предусмотрено в таких языках программирования, как Oberon, Modula-3, Smalltalk, Scheme, ML и Haskell.

Различают три фазы управления кучей:

1. *Начальное распределение.* Изначально вся куча рассматривается как один блок свободной памяти. Когда запрашивается блок из *N* слов, *указатель кучи* перемещается на *N* слов, а его начальное значение возвращается в качестве указателя на размещенную запись активации. Этот процесс может повторяться многократно.
2. *Восстановление памяти.* После того как запись активации перестала быть нужной, память из-под нее должна быть восстановлена для повторного использования. Восстановление памяти реализуется механизмом сбора мусора.
3. *Уплотнение и повторное использование.* Восстановленная память может быть сразу же повторно использована, но может потребоваться и процедура ее уплотнения — построение более крупных блоков свободной памяти из маленьких.

Рано или поздно указатель кучи достигнет конца блока, составляющего кучу. Теперь следует повторно использовать ту часть памяти, которая на данный момент не задействована под необходимые записи активации. Существует два способа повторного использования этой памяти:

- анализировать список свободного пространства, проводя в нем поиск блока подходящего размера и возвращая после выделения памяти оставшуюся часть блока снова в список свободного пространства;
- уплотнить свободное пространство, переместив все активные элементы к одному краю кучи, оставляя при этом все свободное пространство в виде единого блока и переустанавливая указатель кучи на начало этого блока.

Обсудим каждый из этих способов.

Повторное использование свободного пространства

При получении запроса на область размером *N* слов выполняется поиск в списке свободного пространства кучи. Используется один из возможных подходов.

Поиск первого подходящего. Если требуется блок размером в *N* слов, список свободного пространства просматривается до обнаружения *первого* блока, состоящего из *N* или более слов, который затем разбивается на блок из *N* слов и остаток, возвращаемый в список свободного пространства.

Поиск наиболее подходящего. Если требуется блок размером в *N* слов, список свободного пространства просматривается до обнаружения *минимального* блока, количество слов в котором больше или равно *N*. Если в этом блоке точно *N* слов, то он распределяется целиком, как единое целое; если же количество слов в нем больше *N*, то он разбивается на две части, а остаток возвращается в список свободного пространства.

Сохранение элементов в списке свободного пространства в порядке возрастания их размеров делает распределение достаточно эффективным. В этом случае надо последовательно просматривать список, пока не будет найден блок требуемого размера.

Уплотнение свободного пространства

Уплотнение памяти выполняется путем сбора мусора.

Процедура сбора мусора состоит из двух этапов — этапа маркировки и собственно этапа сбора:

1. *Маркировка элементов*. В исходном состоянии каждый активный элемент кучи содержит *бит мусора*, установленный в положение «включен». Алгоритм маркировки по *специальному условию* проверяет активность элемента и, если она подтвердилась, переводит этот бит в положение «выключен».
2. *Сбор элементов*. Здесь последовательно просматриваются все элементы кучи. Все элементы, у которых бит мусора находится в положении «включен», рассматриваются как мусор и возвращаются в список свободного пространства. После завершения сбора биты мусора оставшихся элементов вновь возвращаются в положение «включен». Тем самым элементы готовятся к новому циклу сбора мусора.

УСЛОВИЕ, КОТОРОЕ ПРОВЕРЯЕТ АЛГОРИТМ МАРКИРОВКИ

> Любой активный элемент должен быть достижим по цепи указателей, начинающейся вне кучи.

Конечно, для обеспечения возможности такой проверки алгоритм маркировки должен быть способен:

- обнаруживать все указатели вне кучи, указывающие на элемент внутри нее;
- определять все поля внутри активного элемента кучи, которые содержат указатели на другие элементы.

Сложности проявляются и на этапе сбора. Связаны они с переменным размером элементов. Где заканчивается один элемент и начинается другой? Без этой информации просто невозможно выполнить сбор мусора.

Самым простым решением считается хранение в первом слове каждого активного и неактивного элемента *идентификатора его длины*. Этот идентификатор указывает на количество слов в i-м блоке. При наличии идентификатора длины в каждом элементе кучи достаточно анализировать только первое слово.

С уплотнением свободного пространства тесно связана проблема фрагментации кучи, к обсуждению которой мы переходим.

Фрагментация памяти в куче

Вспомним, что заполнение кучи начинается с одного большого блока свободного пространства. В ходе вычислений этот блок постепенно разбивается на меньшие блоки посредством процедур выделения памяти, ее восстановления и повторного использования. При этом блоки свободного пространства расщепляются на все более

мелкие области. В конце концов, наступит момент, когда менеджер распределения памяти не сможет удовлетворить запрос на блок из *N* слов, поскольку не окажется ни одного достаточно большого блока, хотя в целом список свободного пространства содержит гораздо больше, чем *N* слов свободной памяти. В итоге, из-за нехватки свободной памяти выполнение программы останавливается задолго до того, когда на самом деле исчезнут ресурсы памяти. Таким образом, возникает потребность дефрагментации кучи: объединения свободных блоков в блоки большего размера.

Возможны два подхода к дефрагментации:

1. *Частичная дефрагментация*. Если активные блоки кучи *нельзя* перемещать (или перемещение обходится слишком дорого), то можно объединять только смежные блоки из списка свободного пространства.
2. *Полная дефрагментация*. Если активные блоки *можно* перемещать, тогда все активные блоки перемещаются в один конец кучи, а все свободное пространство оказывается сосредоточенным на другом конце, в одном непрерывном блоке. Полная дефрагментация подразумевает, что при перемещении активного блока все указатели на него модифицируются нужным образом и способны указать на его новое местоположение.

Размещение и освобождение в стеке

Использование стека — это традиционный способ для размещения записи активации. Он применяется в языках Algol 60, C, Pascal и вообще в большинстве императивных языков.

Стековая дисциплина обеспечивает эффективное повторное использование памяти. Память для локальных переменных выделяется при начале активации и освобождается при завершении активации. Когда `P` вызывает `Q`, а `Q` вызывает `R`, память используется в стиле LIFO.

Покажем стек, который растет вниз по мере установки активаций для `P`, `Q`, `R` и сокращается по мере возврата управления из `R` в `Q` и из `Q` в `P` (рис. 14.10).

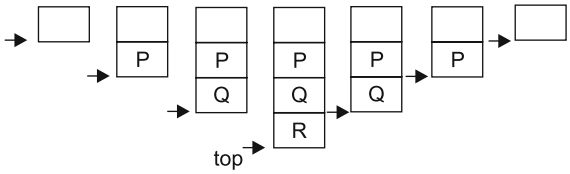


Рис. 14.10. Динамика стека при хранении записей активации

Стек можно применять, если память под переменные не нужна после окончания активации и поэтому может быть освобождена. Это накладывает на программы определенные ограничения.

Дисциплина стека мотивирует ограничения на обработку параметров и результатов подпрограмм. Императивные языки требуют — все подпрограммы объявляются в исходном тексте программы. Не разрешено определять подпрограммы динамически, в период выполнения. В функциональных языках динамические объявления очень нужны, поэтому в чистом виде стек здесь использовать нельзя.

Размещение статических переменных в период компиляции

Локальные переменные ограничены собственной отдельной памятью для каждой активации. Исключение должно быть сделано для локальных переменных, которые объявлены как статические (`static`). Дело в том, что статические переменные сохраняют свои значения между активациями, они используются всеми активациями определенной подпрограммы.

Время жизни статической переменной — весь период вычислений. Она сохраняет свое значение при переходе от одной активации к другой. В языках С и С++ объявление статических переменных начинается со слова `static`. Например:

```
static int count = 0;
```

Память под переменную `count` нельзя уничтожать при выходе из подпрограммы, в которой она объявлена.

Управление подпрограммами в языке С

Язык С спроектирован таким образом, что в записях активации достаточно только связи управления. Связи доступа здесь не нужны, так как запрещено вложение подпрограмм друг в друга. Доступ к объявлению имени здесь возможен или из самой подпрограммы, или из программы вне всех подпрограмм. Таким образом, здесь имеются только две категории объявлений данных:

- глобальные объявления, они доступны всем подпрограммам;
- локальные объявления, которые доступны лишь одной-единственной подпрограмме.

Схема распределения памяти для работающей С-программы представлена на рис. 14.11.

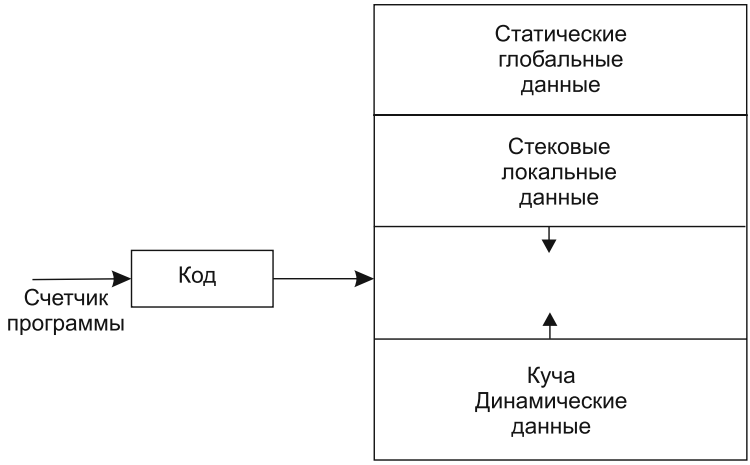


Рис. 14.11. Распределение памяти для программ на языке С

Счетчик программы позволяет отслеживать поток управления через исходный код программы. Этот код в период выполнения не меняется. Глобальные данные хранятся в статически выделенной памяти. Локальные данные хранятся в стеке, который растет в начале активации и сжимается в ее конце. В верхнем секторе памяти располагается куча, которая хранит динамические данные, размещаемые и удаляемые с помощью библиотечных подпрограмм.

Память под локальные переменные выделяется в тот момент, когда поток управления входит в конкретный блок (составной оператор, содержащий объявления), и освобождается, когда управление покидает блок. Это означает, что блоки могут быть реализованы формированием для каждого блока фрейма в стеке: размещение и освобождение фрейма связано с прохождением блока.

Переменные в несвязанных блоках могут разделять одну и ту же память стека, так как они никогда не появляются одновременно.

Формат записи активации в С показан на рис. 14.12.

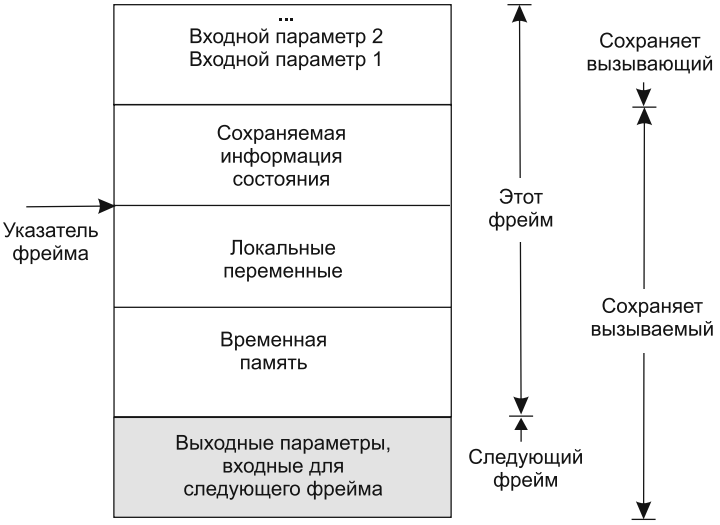


Рис. 14.12. Формат записи активации подпрограммы на языке С

Поле для сохраняемой информации состояния включает связь управления, которая указывает на запись активации вызывающего модуля. Необходимость связи управления обусловлена организацией стека — размер записи активации изменяется (от подпрограммы к подпрограмме). Для подпрограммы `P` все записи активации имеют одинаковый размер. Размер записей активации другой процедуры `Q` зависит от кода для `Q` и в общем случае отличен от размера для `P`.

Работу по размещению и заполнению полей фрейма вызываемого модуля выполняют как вызывающий, так и вызываемый модуль. Собственник фрейма, вызываемый модуль, определяет размеры всех полей, кроме поля для входных параметров. Если вызываемый модуль имеет переменное количество аргументов, то размер поля входных параметров меняется от вызова к вызову.

С-компилятор гарантирует, что при вызове подпрограммы происходят следующие действия:

1. Язык C использует передачу параметров по значению. Поэтому вызывающий модуль вычисляет фактические параметры оператора вызова и размещает их значения в записи активации вызываемого модуля. При вызове `f(n - 1)` переменная `n` локальна для вызывающего модуля, поэтому вычисление фактического параметра `n - 1` — это обязанность вызывающего модуля.
2. Сохраняется информация, требуемая для повторного выполнения вызывающего модуля. Эта информация включает адрес, на который должно быть передано управление при завершении вызова подпрограммы.
3. Вызываемый модуль выделяет место под свои локальные переменные. Резервируется временная память под промежуточные результаты для вычисления выражения. Выражение `(a + b) * c` реализуется следующей последовательностью операторов:

   `t1 := a + b; t2 := t1 * c;` где `t1` и `t2` — имена, генерируемые компилятором. Значения переменных с такими именами сохраняются во временной памяти.
4. Выполняется тело вызываемого модуля. В ходе этого выполнения возможны дополнительные активации. Для вызова, который может произойти в ходе активации, используется разделяемое поле для выходных параметров. Эти выходные параметры становятся для следующего модуля входными параметрами. Размер поля для вызова `P(x, y)` отличается от размера для вызова `Q( )`.
5. Управление возвращается вызывающему модулю. Возвращаемая величина помещается в то место, где ее может найти вызывающий модуль. Восстанавливаются значения машинных регистров, чье состояние запоминалось в начале вызова. Далее управление передается на сохраненный адрес возврата. Стек освобождается так, что последующие активации будут правильно позиционироваться в стеке.

В период выполнения известен формат записи активации, но неизвестна ее позиция в памяти. После помещения фрейма в стек его память доступна при помощи указателя фрейма. Поэтому значения переменных в активации могут быть доступны с помощью следующих адресов:

```
указатель фрейма + смещение
```

где смещение вычисляется в период компиляции.

Представленный на рис. 14.12 формат допустим и для процедур с переменным количеством параметров, например для подпрограммы `printf( )`. Такие подпрограммы могут быть обработаны с использованием первого входного параметра (ближайшего к указателю фрейма) для описания оставшихся параметров. Таким образом, первый параметр находится в известном месте, далее следует сохраняемая информация состояния, другие параметры могут быть доступны с помощью описания в первом параметре.

Управление подпрограммами в языке Pascal

Допустим, что память разделяется на области для кода, статических данных, стека периода выполнения и кучу. Каждая подпрограмма имеет запись активации с полями для параметров, сохраняемой информации состояния, локальных переменных, генерируемых компилятором временных результатов.

Формат фрейма для P-компилятора языка Pascal иллюстрирует рис. 14.13.

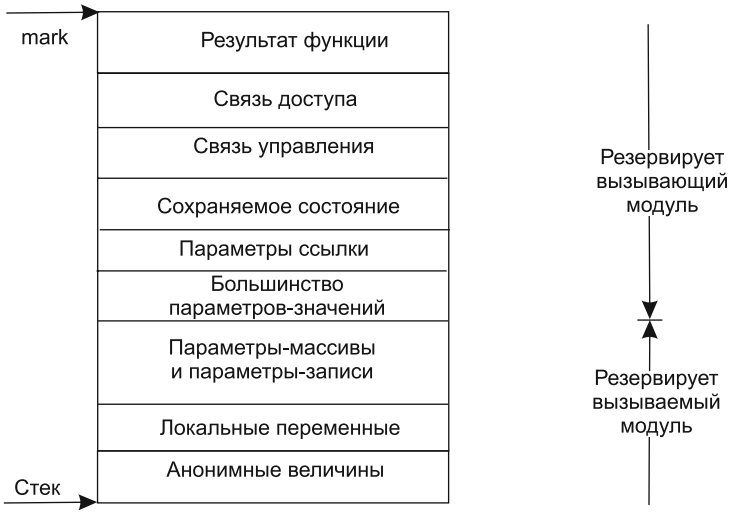


Рис. 14.13. Формат записи активации подпрограммы на языке Pascal

Здесь есть поля для двух видов связей: связи управления и связи доступа.

Связь управления (динамическая связь) указывает на запись активации вызывающего модуля. Она соответствует метке `mark` в начале фрейма.

Связь доступа (статическая связь) указывает на самую новую запись активизации статически объемлющего блока (это блок, в тексте которого содержится описание вызываемого модуля).

Доступ к нелокальным переменным проиллюстрируем с помощью следующей программы:

```
program M;
    procedure P;
        var  x, y, z;
        procedure Q;
            procedure R;
            begin (* R *)
                …
                z := P; …
            end R;
        begin (* Q *)
            …
            y := R; …
        end Q;
        begin (* P *)
        …
        x := Q; …
    end P;
Begin (* M *)
    …
    P;
    …
End M.
```

Конечно, в реальной программе такое использование переменных x, y и z маловероятно. Здесь вхождения нелокальных переменных y в Q и z в R должны ссылаться на их объявления в объемлющей процедуре P. Поэтому в период выполнения в записи активизации для P появляются области размещения y и z.

Зарисуем состояния стека периода выполнения, соответствующие выполнению этой программы (рис. 14.14). Как обычно, отдельные активизации процедур будем различать за счет использования индексов. Договоримся, что связи управления рисуем слева, а связи доступа — справа от прямоугольников активаций.

После передач управления $M \to P \to Q \to R$ стек принимает состояние, показанное на рис. 14.14, *а*.

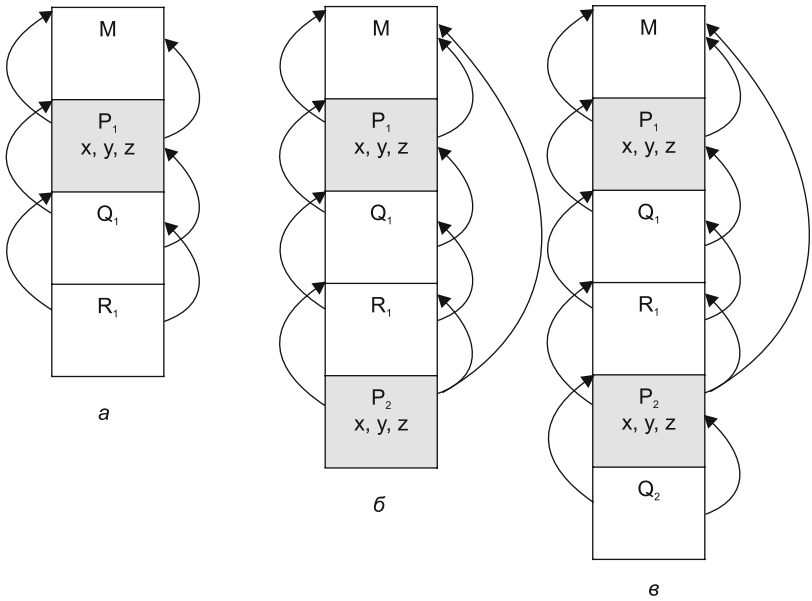


Рис. 14.14. Состояния стека периода выполнения программы M

Это случай, когда каждая вызываемая процедура лексически вложена в вызывающую процедуру (то есть текст вызываемой процедуры расположен внутри текста вызывающей процедуры). Поэтому и связи управления, и связи доступа указывают от вызываемого модуля к вызывающему модулю. Связь управления указывает на начало, помеченное как mark, записи активации вызывающего модуля. Связь доступа указывает на поле связи доступа в записи активации вызывающего модуля.

После вызова из R процедуры P стек принимает состояние, отображенное на рис. 14.14, *б*. Это значит, что активация R_1 делает рекурсивный вызов P_2. Управляющая связь из P_2 указывает на модуль R_1, вызвавший его; связь доступа указывает на M, наиболее новую активацию лексически объемлющего блока.

Нужны ли связи доступа? Да, нужны. Учтем, что процедура P объявлена внутри M. На рис. 14.14, *б* показаны две активации P. Так как P может быть вызвана рекурсивно, в период компиляции невозможно предсказать количество ее активаций. Поэтому в период компиляции не могут быть предсказаны относительные позиции

фреймов для M и активаций для P. Связи доступа из активаций P_1 и P_2 *необходимы* для нахождения фрейма для M в период выполнения.

Исследуем, как используются связи доступа, а затем обсудим, как они сохраняются. В примере, рассмотренном выше, каждая активация P получает свои собственные локальные переменные x, y и z. Таким образом, в активации Q_1 нелокальная переменная y ссылается на место размещения в P_1, в активации Q_2 нелокальная переменная y ссылается на место размещения в P_2. Аналогично z в активации R_1 ссылается на место размещения двух связей доступа возле P_1, а z в активации R_2 ссылается на место размещения двух связей доступа возле P_2.

Количество связей, необходимых для доступа к нелокальным переменным, вычисляют в период компиляции с помощью высоты вложенности блока. Высоту вложенности блока программы считают равной единице: $h_м = 1$. Высота вложенности вложенного блока на единицу больше высоты вложенности объемлющего блока.

ПРИМЕНЯЕТСЯ СЛЕДУЮЩЕЕ ПРАВИЛО

> Высота вложенности переносится от программ к модулям: высота вложенности программы равна 1, а высота вложенности вложенного блока на единицу больше, чем высота вложенности объемлющего блока.

Для нашего примера: $h_P = 2$, $h_Q = 3$, $h_R = 4$.

Расстояние между двумя блоками — это разница между их высотами вложенности:

Расстояние $2 \rightarrow 1 = h_2 - h_1$

В общем случае адрес для нелокального вхождения имени x использует два компонента:

1. Количество пересекаемых связей доступа. Оно равно расстоянию между блоком с нелокальным вхождением x и блоком, содержащим ближайшее объявление x.
2. Относительный адрес x в достигнутой записи активации (смещение относительно позиции связи доступа во фрейме).

Таким образом, нелокальная переменная может быть доступна с помощью известного количества связей доступа и известного смещения. Здесь слово «*известный*» означает «*известный в период компиляции*». Локальные же переменные находятся в известной позиции текущей записи активации.

Начальная установка связей доступа

При вызове подпрограммы для начальной установки этих связей требуется знать расстояние между блоками. Например, расстояние между блоками R и M равно:

$h_R - h_м = 4 - 1 = 3$

Напомним, что в примере из R вызывается P, а в M объявляется P.

Кроме того, здесь число «три» — это количество связей доступа от R_1 до записи активации для M.

Более подробно, когда R_1 вызывает P_2, выполняется следующее:

1. Для нахождения записи активации для M следуют вдоль трех связей доступа от R_1.
2. Устанавливают связь доступа в P_2 для указания на M.

Вызов и возврат подпрограммы

Когда подпрограмма вызывается с использованием представленного формата записи активации, выполняются следующие действия:

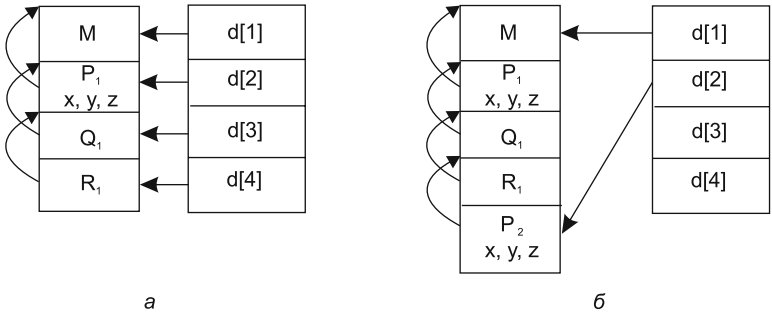
1. Вызывающий модуль размещает поля для результата функции, связей управления и доступа, а также сохраняемой информации состояния (она нужна для возобновления выполнения вызывающего модуля).
2. Связь управления в вызываемом модуле указывает на начало фрейма в вызывающем модуле.
3. Если вызываемый модуль объявлен внутри вызывающего модуля, то связь доступа указывает на связь во фрейме вызывающего модуля, в противном случае связь доступа в вызываемом модуле определяется с использованием высот вложенности вызывающего модуля и вызываемого модуля. Если разница между их высотами вложенности равна `d`, то связь доступа вызываемого модуля будет указывать на фрейм, который находится по `d` связям доступа от фрейма вызываемого модуля.
4. Сохраняется информация, требуемая для повторного запуска вызывающего модуля (как минимум это адрес, по которому следует вернуть управление после завершения вызова).
5. Вызывающий модуль формирует поле для всех параметров-ссылок, адресную информацию о массивах и записях, передаваемых по значению, и все параметры-значения. Это поле должен заполнить вызывающий модуль, так как имена в списке фактических параметров используют связь доступа в вызывающем модуле.
6. Управление передается в вызываемый модуль. Вызываемый модуль формирует поле для массивов и записей, передаваемых по значению. Он копирует значения в это поле. Эта работа может выполняться вызывающим модулем. Использование двух полей для параметров и разделение работы по установке этих полей конкретизируются для данной реализации.
7. Вызываемый модуль резервирует место под локальные переменные и место под временную память для промежуточных результатов в ходе вычисления выражений.
8. Выполняется тело вызываемого модуля. При выполнении тела возможны дополнительные активации.
9. Управление возвращается вызывающему модулю. Возвращаемое значение помещается в поле для результата. Управление передается по сохраненному адресу. Стек текущей активации освобождается, так что последующие активации правильно позиционируются в стеке.

Дисплеи для быстрого доступа к информации

Дисплеи предлагают технику оптимизации, обеспечивающую быстрый доступ к нелокальным переменным. Дисплей — это глобальный массив `d` из указателей на записи активации, индексированных лексической высотой вложения.

Элемент массива `d[i]` всегда указывает на самую новую активацию блока с высотой вложенности `i`. В действительности элементы дисплея являются связями доступа, которые получают следованием по цепочке связей доступа от текущей записи активации к записи для главной программы. Количество элементов дисплея известно в период компиляции (из лексической вложенности в исходном тексте программного кода).

Покажем дисплей как массив `d[1..4]`, в котором `d[1]` указывает на запись активации `M`, `d[2]` указывает на P_1 и т. д. — рис. 14.15, *а*.



а *б*

Рис. 14.15. Элементы дисплея, индексированные высотой вложенности

На рис. 14.15, *б* рассматриваются только элементы `d[1]` и `d[2]`, так как P_2 может ссылаться лишь на имена, объявленные в объемлющих блоках. При возврате из P_2 в R_1 дисплей должен восстановить свое состояние, показанное на рис. 14.15, *а*.

С помощью дисплея нелокальная переменная `x` находится так:

1. Для поиска записи активации, содержащей `x`, применяют лишь возможности дисплея. Если наиболее вложенное объявление `x` имеет высоту вложения `i`, тогда `d[i]` указывает на запись активации, содержащую область размещения `x`.
2. Для поиска области размещения `x` используется только относительный адрес внутри записи активации. Элемент дисплея `d[i]` указывает на предопределенную позицию в записи активации, например на начало записи активации. В период компиляции область размещения `x` может представляться как смещение от предопределенной позиции.

При вызове процедуры `P` должна быть произведена начальная установка глобального дисплея (для того чтобы активация `P` могла его использовать). Применяется следующая методика: вызывающий модуль сохраняет весь массив-дисплей в собственной записи активации. Далее вызываемый модуль может использовать глобальный массив-дисплей в собственных целях.

Размер массива равен максимальной высоте вложенности в программе (обычно это небольшая величина). Для нашего примера эта высота равна `4`. Когда управление возвращается вызывающему модулю, он восстанавливает глобальный дисплей из массива, сохраненного в своей записи активации.

Эта методика позволяет корректно обрабатывать подпрограммы, которые передаются как параметры, но увеличивает накладные расходы для вызова подпрограмм, поскольку требуется сохранять весь массив.

Рассмотрим альтернативную методику, которая применяется, когда подпрограммы не передаются как параметры, но может работать и при их наличии.

Основная идея заключается в следующем: когда происходит активация с высотой вложенности `i`, следует сохранять только `d[i]`. Старое значение `d[i]` сохраняется в записи активации вызывающего модуля.

Усложнение для подпрограмм-параметров состоит в том, что параметр требует установки своего лексического окружения для всего дисплея.

Альтернативную методику использования дисплея иллюстрирует рис. 14.16.

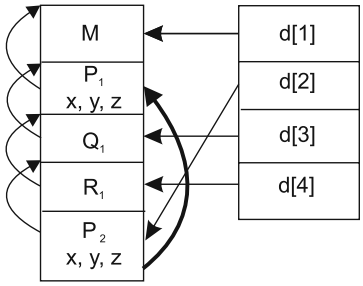


Рис. 14.16. Альтернативная методика использования дисплея

Здесь в R_1 есть вызов P_2. Так как P_2 имеет высоту вложенности 2, элемент дисплея `d[2]` указывает на P_2. Старое значение `d[2]` сохраняется в записи активации для P_2. Это показано утолщенной линией справа от прямоугольника. Элементы дисплея `d[3]` и `d[4]` недоступны. Они остаются на месте, до использования при возврате из P_2 в R_1.

Фрагмент обслуживания дисплея из шагов вызова подпрограммы имеет вид:

1. Сохранить `d[i]` в записи активации для вызывающего модуля с высотой вложенности `i`.
2. Сделать `d[i]` указателем на вызываемый модуль.

Обслуживание дисплея в ходе возврата из подпрограммы состоит в восстановлении `d[i]` с помощью величины, сохраненной в вызванном модуле.

Контрольные вопросы и упражнения

1. Предположим, что некий язык позволяет при объявлении локальных переменных задавать их начальные значения, как, например, в следующем объявлении на языке Ada

   ```
   X: integer := 50;
   ```

   которое инициализирует переменную `X`, присваивая ей значение `50`. Объясните два значения, которые может иметь эта инициализация в случаях:

   - если локальные переменные сохраняются в промежутках между обращениями к подпрограмме;
   - если локальные переменные уничтожаются между вызовами подпрограмм.

2. В языках Pascal и Ada допускается использовать указатели совместно с операцией `new` для создания новых объектов данных. Их применение приводит к генерации мусора во время выполнения программы. Объясните, в чем состоит причина этого явления и как минимизировать такую генерацию.
3. Возьмите какую-либо написанную вами программу на языке, который использует правила статической области видимости, и для каждой содержащейся в ней подпрограммы перечислите имена, входящие в ее среду:
   - локальных ссылок;
   - нелокальных ссылок;
   - глобальных ссылок и
   - предопределенных ссылок.

   Затем решите обратную задачу: для имени, объявленного в каждой из этих сред ссылок, перечислите:
   - объявления подпрограмм в его статической области видимости;
   - активации подпрограмм в его динамической области видимости.
4. Рассмотрим следующую программу, написанную на языке, в котором используются правила статической области видимости:

```
program main(input,output);
    var i,j,k,m: integer;
    procedure Q(var i: integer; m: integer);
        begin
            i:=i+k;
            m:=j+1;
            writeln(i,j,k,m)
        end;
    procedure P(var i: integer; j: integer);
        var k: integer;
        begin
            k:=4;
            i:=i+k;
            j:=j+k;
            Q(i,j)
        end;
    begin
        i:=1;
        j:=2;
        k:=3;
        P(i,k);
        writeln(i,j,k)
    end.
```

   Заполните следующую таблицу для каждого оператора `writeln` в предположении, что передача параметров осуществляется указанным способом.

Способ передачи параметров	**i**	**j**	**k**	**m**
Язык Pascal, как записаны в программе				
Все параметры передаются по ссылке				
Все параметры передаются по значению				
Все параметры передаются по значению-результату				

5. Операция выделения памяти `malloc` языка C часто реализуется как обращение к программе распределения памяти, встроенной в операционную систему. Операционная система организует центральную кучу, используемую всеми программами, а операции `malloc` и `free`, вызываемые из C-программы, соответственно выделяют и освобождают память в этой центральной куче. Поскольку операция `malloc` требует обращения к операционной системе, этот процесс может занять много времени. Альтернативным вариантом реализации операции `malloc` является использование локальной кучи в адресном пространстве каждой программы. В таком случае использование этой операции обходится значительно дешевле. (Операция `new` в языке Pascal реализована таким же образом.) Предположим, ваша программа требует достаточно частого использования операции `malloc` для выделения небольших блоков памяти из кучи. Исследуйте вашу локальную реализацию C и определите, какова в этой реализации стоимость использования для указанных целей операции `malloc`.

Глава 15
Абстрактные типы данных

В этой главе излагаются вопросы, связанные с крупноблочным строительством модулей программы. Вначале обсуждается специфика инкапсуляции — занесения данных и подпрограмм в единую капсулу, итогом которой становится абстрактный тип данных (АТД). Функциональные возможности АТД превышают возможности одиночной подпрограммы. Далее приводятся примеры создания и использования обычных АТД (в различных языках программирования). Область применения АТД увеличивается за счет параметрического полиморфизма. Таким способом разрабатываются родовые АТД, которые также поясняются примерами. Следующий шаг в этом же направлении приводит нас к более крупным контейнерам (физическим контейнерам), в каждый из которых можно положить несколько типов данных. После рассмотрения этого шага глава переключает внимание читателя на пространства имен, то есть логические контейнеры, поддерживающие параллельную и независимую разработку очень больших программ.

Абстракция процесса

Абстракция — это упрощенное представление реальной сущности с важными для данного контекста характеристиками. При построении абстракции второстепенные характеристики сущности игнорируются. Абстракция упрощает рассмотрение сущности, выделяя существенные свойства и игнорируя второстепенные свойства.

В современных языках программирования различают абстракции процессов и абстракции данных [12, 13, 17, 24, 29, 31, 32]. Исторически первыми появились *абстракции процессов* [82]. Они обслуживали императивный стиль программирования, в котором основными действующими лицами программы являлись действия. Все подпрограммы (процедуры и функции) — абстракции процессов. Чем они удобны? Дело в том, что описание программы в терминах подпрограмм является способом существенного упрощения самого процесса программирования.

Например, для сортировки числовых объектов можно использовать оператор вызова процедуры `sort_double (list, size)`. Этот вызов — абстракция действительного процесса, чей алгоритм не определяется. Вызов не зависит от алгоритма, реализуемого в процедуре.

Существенными характеристиками абстракции `sort_double` являются:

1) имя сортируемого массива;
2) тип его элементов;
3) длина массива;
4) тот факт, что результат — это отсортированный массив.

В данном случае конкретный алгоритм, который реализует процедура, — это характеристика, несущественная для пользователя. А ведь на изучение и составление алгоритма пришлось бы потратить много усилий!

Абстракция процесса имеет решающее значение для программ. Возможность абстрагироваться от деталей алгоритмов в подпрограммах значительно упрощает разработку и понимание больших программ.

С появлением объектно-ориентированного подхода стали говорить о том, что абстракций процесса недостаточно для крупноблочного строительства программ. Необходимым инструментом стали абстракции данных, покрывающие более крупные фрагменты программного кода. Об абстракциях данных пойдет речь в последующих разделах. Заметим, что абстракция процесса напрямую используется в абстракции данных, так как важной частью абстракции данных являются ее операции.

Инкапсуляция и абстракция данных

При очень больших размерах программу приходится организовывать с помощью синтаксических контейнеров, включающих группировки логически связанных операций и данных.

Инкапсуляция — это такое размещение в контейнере данных и подпрограмм для их обработки, при котором доступ к данным возможен только с помощью подпрограмм. Иначе говоря, инкапсуляция обеспечивает *принцип сокрытия информации* [65, 78].

ПРИНЦИП СОКРЫТИЯ ИНФОРМАЦИИ (ИНФОРМАЦИОННОЙ ЗАКРЫТОСТИ), ЛЕЖАЩИЙ В ОСНОВЕ ПРОЕКТИРОВАНИЯ АБСТРАКЦИЙ, УТВЕРЖДАЕТ

> каждый программный компонент, определяемый программистом, должен насколько возможно скрывать от пользователей информацию о своем внутреннем устройстве.

Когда информация инкапсулирована в абстракцию, это означает, что пользователь:

- не нуждается в скрытой информации для использования абстракции;
- даже при желании не имеет возможности как-нибудь манипулировать скрытой информацией. Ему это просто не позволено.

Например, целочисленный тип данных, встроенный в языки программирования, не только скрывает детали представления целых чисел в памяти, но и инкапсулирует это представление таким образом, что программист не имеет возможности манипулировать отдельными битами в представлении целого.

Механизм инкапсуляции является естественным продуктом эволюции понятия «тип данных».

Первоначально тип данных задавался как множество значений, которые может принимать конкретная переменная. Типы данных непосредственно связывались с отдельными переменными, так что каждое объявление определяло имя переменной и ее значения. Если в программе применялось несколько одинаковых массивов, то выполнялось следующее:

- каждый массив определялся отдельно;
- для каждого определения повторялось полное описание массива.

В конце шестидесятых годов Н. Вирт обобщил понятие типов данных до общей формулировки. По Вирту, определение типа задавало структуру объекта данных вместе с возможными значениями этой структуры. Для объявления объекта данных, принадлежащего конкретному типу, требовалось указать только имя переменной и название этого типа.

В семидесятые годы понятие типов данных получило дальнейшее развитие, оно переросло представление о типе просто как о множестве объектов данных с возможными значениями. В понятие типа данных был добавлен набор операций над этими объектами.

Для целочисленных и вещественных типов данных в языке предусматривались средства объявления переменных этих типов и набор операций над ними. Эти операции являлись единственным инструментом программиста для манипулирования вещественными и целыми числами.

Следовательно, представление целых и вещественных чисел в языке программирования было эффективно инкапсулировано, то есть скрыто от программиста. Программист может применять целые и вещественные числа, не вникая в детали их реализации и представления в памяти. Программист видит только название типа и список определенных для него операций, являющийся инструментарием для обработки объектов данных этого типа.

Все эти средства и возможности были заданы для типов данных, встроенных в язык программирования. Естественно, что следующим шагом эволюции стало обеспечение возможности расширения набора типов для языка, распространения механизма инкапсуляции на определяемые программистом типы данных. Так сформировалось понятие абстракции данных.

АБСТРАКЦИЯ ДАННЫХ ОПРЕДЕЛЯЕТ

- множество объектов данных с конкретным набором значений;
- набор операций над объектами данных;
- такую инкапсуляцию этих объектов и операций, при которой пользователь нового типа данных может манипулировать объектами данных только с помощью заданных операций.

В свою очередь, само описание абстракции данных в целом должно быть так инкапсулировано, чтобы пользователю было достаточно знать лишь название типа и смысл доступных операций.

Абстрактные типы данных

Абстрактный тип данных (АТД) — это синтаксический контейнер, который включает определение данных только одного типа и подпрограммы, которые обеспечивают операции для этого типа.

АТД полностью реализует идею инкапсуляции. Детали типа скрыты от пользователей его капсулы. Программные модули — пользователи — могут объявлять экземпляры АТД, называемые объектами, хотя действительное и полное представление АТД скрыто от них.

Более конкретно: структура объекта, а также тела подпрограмм скрыты от пользователя. К объектам применимы только те операции, которые описаны в АТД.

Одна из целей введения АТД — борьба со сложностью программирования.

Напомним, что понятие АТД, по крайней мере в терминах встроенных типов языка программирования, — это не современная разработка. Все встроенные типы данных (даже в языке Fortran) являются абстрактными типами данных, правда, они редко так называются.

Например, рассмотрим числовой тип данных с плавающей точкой (`float`). Этот тип обеспечивает средства для создания переменных с плавающей точкой и набор операций для их обработки. Кроме того, тип реализует ключевое понятие абстракции данных — сокрытие информации. Действительный формат переменной с плавающей точкой скрыт от пользователя. Пользователю доступны только операции, обеспечиваемые языком. Пользователю запрещают создание новых операций над данными типа, запрещают обработку частей действительного представления объектов с плавающей точкой.

Более современно понятие абстрактных типов данных, определяемых программистом.

АТД, ОПРЕДЕЛЯЕМЫЙ ПРОГРАММИСТОМ, УДОВЛЕТВОРЯЕТ ДВУМ УСЛОВИЯМ

1. Представление (определение) типа и операции над объектами типа содержатся в едином синтаксическом модуле. Другие программные модули могут лишь создавать переменные этого типа.
2. Представление экземпляров типа недоступно пользователям. К объектам применимы только те операции, которые обеспечены определением типа.

Программные модули, которые используют конкретный АТД, называются клиентами этого типа.

Информационная закрытость АТД обеспечивает следующие преимущества:

1. Код клиента не зависит от деталей реализации АТД. Это значит, что внутреннее представление АТД можно менять без модификации клиента.
2. Возрастает надежность программ. Так как клиенты не могут изменять объекты непосредственно, а только через допустимые операции, возрастает целостность объектов.

Пример. Предположим, что АТД — это стек со следующими абстрактными операциями.

Операция	Обязанность
create (stack)	Создание и, возможно, инициализация объекта-стека
destroy (stack)	Освобождение памяти из-под объекта-стека
empty (stack)	Функция возвращает `True`, если стек пуст, и `False` в противном случае
push (stack, element)	Вталкивание определенного элемента в определенный стек
pop (stack)	Удаление верхнего элемента из определенного стека
top (stack)	Возврат копии верхнего элемента из определенного стека

Собственно говоря, содержимое таблицы — это все, что нужно знать клиенту, желающему использовать стек. Заметим, что для создания и удаления стека не нужны детали его реализации. Клиент типа стек может выполнять следующие действия:

```
Program Клиент_Стека;
. . .
create (STK1);               -- создание экземпляра стека
push (STK1, COLOR1);         -- занесение в стек первого элемента
push (STK1, COLOR2);         -- добавление в стек второго элемента
if (not empty (STK1))        -- анализ пустоты стека
    then TEMP := top (STK1); -- получение копии верхнего элемента
    . . .
End.
```

Как видим, программа клиента чрезвычайно проста.

Положим, что оригинальная реализация абстракции стека использовала массив. Позже возникла необходимость замены массива на связный список. Такая замена не влияет на клиента абстракции стека. Главное — оставить неизменными заголовки объявленных операций.

Определение АТД должно обеспечиваться синтаксической конструкцией, которая способна инкапсулировать определения данных и операций абстракции.

Клиенты абстракции должны видеть имя типа и заголовки подпрограмм (в этой конструкции). Это позволит клиентам объявлять переменные абстрактного типа и обрабатывать их значения. Реализация типа должна оставаться тайной. То же справедливо в отношении подпрограмм — заголовки должны быть видимы, а тела скрыты.

Иногда, помимо основных операций АТД, для объектов абстрактного типа разрешаются и встроенные операции языка (сравнение, присваивание и т. д.).

Существует также набор служебных операций, требуемый большинству АТД, он включает:

- конструкторы — инициируют создаваемые объекты (экземпляры АТД);
- деструкторы — освобождают память из-под объектов;
- итераторы — обеспечивают доступ к содержанию объектов по частям, в определенном порядке.

Однако эти операции не универсальны, поэтому должны разрабатываться отдельно для каждого абстрактного типа.

АТД в языке Ada

Удобной синтаксической конструкцией для размещения АТД в языке Ada является пакет.

Каждый пакет состоит из двух частей: спецификации и тела. Спецификация пакета объявляет предлагаемые услуги, а тело содержит реализацию этих услуг.

Спецификация пакета записывается в виде:

```
package <ИмяПакета> is
        <объявления вспомогательных типов, переменных, констант>
        <спецификации процедур и функций>
    private
        <закрытое объявление основного, приватного типа>
end <ИмяПакета>;
```

Спецификация пакета делится на две секции:

- *публичную секцию*, она определяет все данные, вспомогательные типы и подпрограммы, которые известны и видимы для клиентов; здесь же указывается имя закрытого, приватного типа данных, структура которого скрывается от клиентов;
- *приватную секцию*, начало которой обозначено ключевым словом `private`; она содержит полное определение приватного типа данных, которое недоступно клиентам.

Пример спецификации:

```
package ADT_Stack is
    type Stack is private;
    Stack_Error : exception;
    procedure Reset ( The : in out Stack );
    procedure Push ( The : in out Stack; Item : in Integer );
    procedure Pop ( The : in out Stack; Item : out Integer );
private
    Size : Positive := 100
    type Stack_Index is new Integer range 0 .. Size;
    subtype Stack_Range is Stack_Index
                range 1 .. Stack_Index ( Size );
    type Stack_Array is array ( Stack_Range ) of Integer;
    type Stack is record
        Elements : Stack_Array;   -- массив элементов
        Tos : Stack_Index := 0;   -- указатель на вершину
    end record;
end ADT_Stack;
```

Данная спецификация предлагает клиентам один тип данных (клиенту сообщается только имя этого типа — `Stack`), одно исключение и три процедуры. Полное описание приватного типа определено в приватной секции, где описаны также вспомогательные типы и константы.

Тело пакета представляется в виде:

```
package body < ИмяПакета > is
    < объявления локальных переменных, констант, типов >
    < тела процедур и функций >
end < ИмяПакета >;
```

В тело пакета могут входить также дополнительные объекты и типы данных, которые должны быть скрыты. Содержание тела пакета клиентам недоступно.

Пример тела:

```
package body ADT_Stack is
    procedure Reset ( The : in out Stack ) is
    begin
        The.Tos := 0;
                -- установить указатель  в нуль (нет элементов)
    end Reset;
    procedure Push ( The : in out Stack; Item : in Integer ) is
    begin
        if The.Tos /= Stack_Index ( Size ) then
                        -- проверка заполнения
            The.Tos := The.Tos + 1;  -- указатель вверх
            The.Elements (The.Tos) := Item; -- запись элемента
        else
            raise Stack_Error;    -- ошибка переполнения
        end if;
    end Push;
procedure Pop ( The : in out Stack; Item : out Integer ) is
    begin
        if The.Tos > 0 then
            Item := The.Elements (The.Tos ); --счит.верхн.элемен.
            The.Tos := The.Tos - 1;  -- указатель вниз
else
            raise Stack_Error;
                -- ошибка удаления из пустого стека
        end if;
    end Pop;
end ADT_Stack;
```

В качестве примера приведем программу-клиента, работающую с объектами этого стека:

```
with Ada.Text_IO, Ada.Integer_Text_IO, ADT_Stack;
use Ada.Text_IO, Ada.Integer_Text_IO, ADT_Stack;
procedure Main is
    Number_Stack : Stack;  -- объект - числовой стек
    Action : Character;      -- действие
    Number : Integer;  -- обрабатываемое число
begin
    Put_Line ("'+' - Добавить элемент");
    Put_Line ("'-' - Удалить элемент");
    Put_Line ("'x' - Выход");
    New_Line (2);
    loop    -- начало цикла
        begin   -- начало блока
            Put ("Выберите нужный режим: ");
            Get ( Action );
            New_Line;
            Skip_Line;
            case Action is
                when '+' =>
                    Put ("Введите число: "); Get ( Number );
                    Push ( Number_Stack, Number );
                    Put ("Добавлено число = "); Put ( Number );
                    New_Line;
                when '-' =>
                    Pop ( Number_Stack, Number );
                    Put ("Удалено число = "); Put ( Number );
```

```
                    New_Line;
                when 'x' =>  Exit;
                when others =>
                    Put ("Неправильный режим !");
                    New_Line;
            end case;
        exception  -- ловушка исключений
            when Stack_Error =>
                    Put ("Ошибка стека !"); New_Line;
            when Data_Error =>
                    Put ("Это не целое число !");
                    Skip_Line; New_Line;
        end;   -- конец блока
    end loop;  -- конец цикла
end Main;
```

Допустим, что потребовалось расширить функции обработки в `ADT_Stack`, включив следующие дополнительные операции.

Операция	Обязанность
Top	Возвращает верхний элемент из стека, без удаления его из стека
Items	Возвращает текущее количество элементов в стеке

Реализация этих операций требует прямого доступа к приватным элементам `ADT_Stack`. Простое расширение базового пакета вынудило бы перекомпилировать все модули-клиенты, что неудобно. Можно поступить иначе — разместить расширение в дочернем пакете.

Существует два вида дочерних пакетов: публичный пакет, приватный пакет. Рассмотрим использование публичного дочернего пакета. Его спецификация имеет вид:

```
package ADT_Stack.Additions is
    function Top ( The : in Stack ) return Integer;
    function Items ( The : in Stack ) return Integer;
private
end ADT_Stack.Additions;
```

Возможна следующая реализация дочернего АТД:

```
package body ADT_Stack.Additions is
    function Top ( The : in Stack ) return Integer is
    begin
        return The.Elements ( The.Tos );
    end Top;
    function Items ( The : in Stack ) return Integer is
    begin
        return Integer ( The.Tos );
    end Items;
end ADT_Stack.Additions;
```

ПРИМЕЧАНИЕ

Внутри тела дочернего пакета доступен приватный тип родителя, то есть тип Stack. В общем случае дочерний пакет имеет имя Parent.Child, где Parent — имя родителя. Указание контекста на родителя в дочернем пакете не требуется. Объекты родителя прямо видимы без указателя сокращений (use).

Правила видимости дочернего публичного пакета:

1. Считается, что пакет объявлен внутри области определения родителя, но после окончания спецификации родителя (его спецификация как бы «присоединяется» к концу спецификации родителя).
2. Из приватной части и тела дочернего пакета видна приватная часть родителя.
3. Из видимой части дочернего пакета не видна приватная часть родителя. Это запрещает экспорт клиентам скрытых приватных деталей родителя.

Для тестирования дочернего модуля можно применить программу:

```
with Ada.Text_IO, Ada.Integer_Text_IO,
            ADT_Stack, ADT_Stack.Additions;
use Ada.Text_IO, Ada.Integer_Text_IO,
            ADT_Stack, ADT_Stack.Additions;
procedure Main is
    My_Stack : Stack;
begin
    Push (My_Stack, 10);
    Push (My_Stack, 20);
    Put ("Элемент на вершине ");
    Put ( Top (My_Stack)); New_Line;
    Put ("Количество элементов ");
    Put ( Items (My_Stack));
end Main;
```

Таким образом, дочерние пакеты решают следующие проблемы:

- разделение приватного типа по нескольким компилируемым модулям;
- расширение пакета без перекомпиляции клиентов.

Пакет может иметь несколько потомков, которые, в свою очередь, тоже могут обзавестись потомками. Таким путем строится иерархическое дерево, обеспечивающее естественную декомпозицию средств. Потомок может иметь приватную часть. Она видима из его потомков, но невидима для родителя.

АТД в языке C++

В принципе для группировки данных можно применить такую конструкцию, как структура `struct`, но она не обеспечивает информационной закрытости. Целиком и полностью требованиям АТД удовлетворяют такие синтаксические единицы языка C++, как классы. C++ класс — это описание типа данных.

Данные в классе называют *элементами-данными*, а функции в классе называют *элементами-функциями*.

Все экземпляры (объекты) класса разделяют один и тот же набор элементов-функций, но каждый экземпляр получает собственный набор из элементов-данных класса.

Класс может содержать как скрытую от клиентов секцию (*private-секцию*), так и видимую секцию (*public-секцию*).

Различают два способа определения элемента-функции в классе:

- в классе может появиться полное определение (заголовок и тело);
- только заголовок элемента-функции.

При полном определении говорят, что элемент-функция неявно встроена. При вызове такой функции ее код помещается в точку вызова. Если в классе имеется только заголовок элемента-функции, то ее полное определение находится вне класса и компилируется отдельно. Обычно встраивают небольшие элементы-функции (они занимают в клиенте мало места и сокращают время линковки-связывания).

C++ класс может содержать как скрытые, так и видимые элементы. Скрытые элементы помещают в private-секцию, а видимые — в public-секцию.

Public-секция определяет интерфейс для объектов класса. Имеется еще и третья категория видимости (protected), ее обсудим позже.

Конструктор класса имеет то же имя, что и класс. В классе может быть несколько конструкторов. Они отличаются друг от друга профилем параметров. Конструкторы вызываются неявно, при создании объектов класса.

Функция-деструктор вызывается тоже неявно, когда заканчивается время жизни объекта. Динамические элементы объектов удаляются из памяти по оператору `delete`, включаемому в деструктор. Имя деструктора начинается с символа ~, за которым записывается имя класса.

Рассмотрим реализацию АТД-стека с помощью C++ класса:

```
# include <iostream.h> //** подключение станд. библиотеки ввода-вывода
    class Stack  {
    private:    //**скрытые элементы
    int *stack_ptr;
    int   max_len;
    int   tos;
    public:     //** эти элементы видимы клиенту
        Stack ( )   { //** конструктор
            stack_ptr = new int [500];
            max_len = 499;
            tos = -1
        }
        ~ Stack ( )  {delete [ ]  stack_ptr;}; //** деструктор
        void push (int number)  {
            if (tos  = =  max_len)
                cout << "Error in push – stack is full \n";
            else stack_ptr [++tos] = number;
            }
        void pop ( )  {
            if  (tos  = =  -1)
                cout << "Error in pop – stack is empty \n";
            else  tos --;
        }
        int top ( )  {return (stack_ptr [tos]);}
        int empty ( )  {return (tos  = =  -1);}
    }
```

ПРИМЕЧАНИЕ

1. Конструктор использует оператор `new` для размещения в куче массива из 500 целых элементов.
2. Деструктор использует оператор `delete [ ]` для освобождения памяти из-под массива.
3. Поскольку в класс включены тела элементов-функций, все они неявно встраиваются в точки вызова кода клиента.

В private-секции этого класса размещены недоступные для клиента элементы-данные: переменная указателя стека `stack_ptr`, переменная длины стека `max_len` и переменная для верхнего элемента стека `top_ptr`.

Приведем пример программы-клиента, использующей АТД-стек:

```
void main  ( )  {
    int top_one;
    Stack  stk;  // создание экземпляра класса стек
    stk.push (72);
    stk.push (64);
    top_one = stk.top ( );
    stk.pop ( );
    . . .
}
```

Здесь создается объект `stk` — экземпляр АТД `Stack`. В процессе создания неявно вызывается конструктор класса, обеспечивающий размещение стека в куче. Далее в объект посылается первое сообщение `stk.push(72)`, инициирующее выполнение объектом операции `push( )`. В результате в стек заносится число `72`. Формат любого сообщения стандартен:

```
<Имя_объекта>.<Имя_операции >;
```

При достижении закрывающей скобки программы время жизни переменных `top_one` и `stk` заканчивается, поэтому неявно вызывается деструктор для объекта `stk`. В итоге в куче освобождается память из-под стека.

АТД в языке Java

Абстрактные типы данных в языке Java поддерживаются почти так же, как и в языке C++. И все же существует ряд важных отличий. Все объекты здесь размещаются только в куче, а доступ к ним организуется с помощью ссылок. Все методы в языке Java могут быть описаны лишь внутри класса. Тело метода должно записываться вслед за его заголовком. Следовательно, абстрактный тип данных в языке Java полностью размещается внутри одной синтаксической единицы. Внутри Java-класса нет приватной и публичной секций. Уровень доступа (`private` и `public`) задается ключевым словом прямо в определении метода. Аналогичным образом указывается доступность переменной. Если уровень доступа метода или переменной явно не записан, то это означает пакетный доступ, который будет обсуждаться чуть позже.

Рассмотрим определение класса для стека на языке Java:

```
class Stack {
    private int [] refStack;
    private int maxLen, tos;
    public Stack() { // A constructor
        refStack = new int [500];
        maxLen = 499;
        tos = -1;
    }
    public void push(int number) {
        if (tos == maxLen)
            System.out.println("Error in push – stack is full");
```

```
        else refStack[++tos] = number;
    }
    public void pop() {
        if (tos == -1)
            System.out.println("Error in pop – stack is empty");
        else --tos;
    }
    public int top() {
        return (refStack[tos]);
    }
    public boolean empty() {return (tos == -1);}
}
```

Приведем пример класса, который использует стек:

```
public class Client {
    public static void main(String[] args) {
        Stack stk = new Stack();
        stk.push(72);
        stk.push(64);
        System.out.println("number is: " + stk.top());
        stk.pop();
        System.out.println("number is: " + stk.top());
        stk.pop();
    }
}
```

Как видно из приведенных фрагментов, деструкторы классов в языке Java отсутствуют. В них нет необходимости, поскольку здесь работает механизм автоматического неявного сбора мусора.

АТД в языке C#

С одной стороны, язык C# позаимствовал очень многие конструкции из языков C++ и Java. С другой стороны, в него включен ряд новых конструкций. Подобно языку Java, все экземпляры классов C# являются динамическими объектами и размещаются в куче. Для всех классов предопределены стандартные конструкторы (по умолчанию), которые обеспечивают начальные значения данных (`0` для целых типов и `false` для булевых типов). Конечно, пользователь может добавить ряд новых конструкторов, обеспечивающих специфическую инициализацию элементов данных, при этом любая экземплярная переменная, не проинициализированная пользовательским конструктором, получит значение от стандартного конструктора.

Деструкторы в языке C# тоже предусмотрены, однако используются они редко, поскольку применяется автоматическая сборка мусора.

Для создания АТД в языке C# могут быть применены структуры `struct` или классы. Структуры в языке C# существенно отличаются от своих прообразов в языке C++, по сути, они являются облегченными классами. Они могут иметь конструкторы, свойства, методы, поля данных и способны реализовывать интерфейсы, но не поддерживают наследования.

Существенное различие между структурами и классами заключается в том, что структуры являются типами-значениями, а классы — ссылочными типами. Структуры размещаются в стеке, а не в куче. При использовании в

качестве параметров структуры передаются по значению. Все типы-значения в C#, включая все примитивные типы, принадлежат к структурам. Экземпляры структур могут создаваться путем их объявления, подобно экземплярам других предопределенных типов-значений, таких как `int` или `float`. Кроме того, они могут создаваться с помощью оператора `new`, который вызывает конструктор для их инициализации.

Структуры применяют для реализации относительно небольших и простых типов, которые никогда не будут наследоваться. Они удобны для создания объектов, размещаемых в стеке (в противоположность куче).

Три уровня доступа (`public`, `private` и `protected`) в C# указываются тем же путем, что и в языке Java. От языка Delphi C# унаследовал свойства как средства реализации операций `get()` и `set()` (без явного их вызова клиентом). Свойства обеспечивают неявный доступ к определенным приватным данным экземпляра. К примеру, рассмотрим следующий простой класс:

```
public class Observation {
    public int NumberOfDays { //** NumberOfDays – это свойство
        get {
            return number_Days;
        }
        set {
            if(value < 0 || value > 30)
                Console.WriteLine(
                "Value is out of range: {0}", value);
            else
                number_Days = value;
        }
    }
    private int number_Days;
    . . .
}
```

В классе `Observation` определено свойство `NumberOfDays`. Это свойство обеспечивает как метод чтения (`get`), так и метод записи (`set`) для доступа к приватному элементу данных `number_Days`. Допустим, что фрагмент программы клиента имеет вид:

```
Observation ob = new Observation ();
int actualPeriod, oldPeriod;
. . .
oldPeriod = ob.NumberOfDays;
. . .
ob.NumberOfDays = actualPeriod;
```

В этом фрагменте элемент класса `number_Days` рассматривается как переменная с публичным уровнем доступа, хотя на самом деле элемент приватен и доступ к нему возможен только через свойство. Обратите внимание на использование неявной переменной в методе записи. Это механизм, посредством которого ссылаются на новое значение свойства.

Пример со стеком не приводится. Разница между версией на C# и версией на Java минимальна: меняются только методы вывода сообщений и тип возвращаемого результата для метода `empty()` (вместо `boolean` надо поставить `bool`).

АТД в языке Ruby

Язык Ruby поддерживает абстрактные типы данных с помощью классов. Функционально классы Ruby подобны классам в C++ и Java. Класс в Ruby определяется как составной оператор, открываемый зарезервированным словом `class` и закрываемый словом `end`. Имена экземплярных переменных имеют специальную синтаксическую форму: они должны начинаться со знака `@`. Определение экземплярного метода имеет тот же синтаксис, что и у других функций в Ruby: начинается с зарезервированного слова `def` и заканчивается словом `end`. Метод класса отличается от экземплярного метода тем, что к началу его имени добавляется имя класса, за которым следует точка. Например, в классе по имени `Stack` имя метода класса должно начинаться со слова `Stack`. Инициализацию в Ruby выполняют конструкторы. Конструктор не может быть перегружен, в классе допустим только один конструктор.

Классы в Ruby динамичны, в том смысле, что элементы к ним могут добавляться в любое время: достаточно просто включить определения для новых элементов. Более того, даже предопределенные классы языка (например, `String`) могут быть расширены. Например, рассмотрим следующее определение класса:

```
class theClass
    def method1
        . . .
    end
end
```

Этот класс может быть расширен добавлением дополнительного метода `method2` с помощью второго определения класса:

```
class theClass
    def method2
        . . .
    end
end
```

Методы можно также удалять из класса. Достаточно записать новое определение класса, в котором у метода `remove_method` в качестве параметра указывается имя удаляемого метода. В качестве примера удалим метод `method2` из класса `theClass`:

```
class theClass
    remove_method( :method2)
end
```

Здесь параметр `:method2` записан в форме *обозначения*. Обозначение пишется путем присоединения к идентификатору префикса — символа двоеточия. Динамические классы Ruby отчетливо демонстрируют конфликт читабельности с гибкостью. Легкость динамических изменений в классах существенно повышает гибкость языка, но в ущерб читабельности. Для выяснения поведения класса в конкретном месте нужно найти все его определения в программе и рассмотреть их.

Управление уровнем доступа для методов в Ruby тоже имеет динамический характер. По умолчанию для метода задается публичный уровень доступа, но возможен защищенный или приватный уровень.

Уровни доступа всех методов объявляются тремя отдельными методами с именами `public`, `private` и `protected`. Методы доступа могут быть вызваны внутри определения любого класса. Есть два способа вызова этих методов. Без параметров они задают, что все последующие определения методов будут иметь указанный уровень доступа. К примеру, в определении нашего класса методы управления доступом могут использоваться следующим образом:

```
class theClass
    # Сюда помещаются публичные методы
    # Следующие методы будут защищенными
    protected
    # Сюда помещаются защищенные методы
    # Следующие методы будут приватными
    private
    # Сюда помещаются приватные методы
end # закрывающая скобка класса theClass
```

Методы также могут быть вызваны с указанием имени одного или нескольких методов в качестве параметров (в виде обозначений или строк). При вызове таким способом они изменяют уровень доступа указанных методов. Принято, что объявление уровня доступа должно следовать после определения метода:

```
class theClass
    . . .
    def method3
    . . .
    end
    . . .
    def method5
    . . .
    end
    private :method3, :method5 . . .
end # закрывающая скобка класса theClass
```

В этом примере приватный уровень доступа задан для методов `method3` и `method5`. Все элементы данных класса считаются приватными, их уровень доступа изменить нельзя. Элементы данных обрабатываются только методами класса, некоторые из которых могут быть методами для обеспечения доступа. Экземплярные данные, доступные с помощью методов, здесь называют атрибутами.

Для экземплярной переменной по имени `@rez` возможны следующие методы чтения и записи:

```
def rez
    @rez
end
def rez = (new_rez)
    @rez = new_rez
end
```

Заметим, что имена методов повторяют имя экземплярной переменной без знака `@`. Имя у метода записи такое же, как у соответствующего метода чтения, но с добавлением знака равенства (=).

Методы чтения и записи могут неявно генерироваться системой в определение класса при включении вызовов `attr_reader` и `attr_writer` соответственно. Параметрами для этих символов служат имена атрибутов:

```
attr_reader :rez, :total
attr_writer :rez
```

Класс стек на языке Ruby имеет следующий вид:

```
class Stack
# Конструктор
    def initialize
        @ref_stack = Array.new(500)
        @max_len = 500
        @tos = -1
    end
# Метод push
    def push(number)
        if @tos == @max_len
            puts "Error in push - stack is full"
        else
            @tos = @tos + 1
            @ref_stack[@tos] = number
        end
    end
# Метод pop
    def pop
        if @tos == -1
            puts "Error in pop - stack is empty"
        else
            @tos = @tos - 1
        end
    end
# Метод top
    def top
        @ref_stack[@tos]
    end
# Метод empty
    def empty
        @tos == -1
    end
end # Закрывающая скобка класса Stack
```

Этот класс определяет структуру стека, который может сохранять объекты *любого типа*. Приведем пример клиента, использующего стек:

```
stk = Stack.new
stk.push(72)
stk.push(64)
puts "number is : #{stk.top}"
stk.pop
puts " number is: #{stk.top}"
stk.pop
```

В этом клиенте выражение #{переменная} преобразует значение переменной в строку, которая затем вставляется в ту строку, где появилось выражение.

В языке Ruby все сущности являются объектами, а массивы считаются массивами ссылок на объекты. Все это делает массив более гибким, чем в случае языков Ada, C++ и Java. Более того, при передаче в конструктор требуемой длины объекты соответствующего класса приобретают возможность иметь любую длину. Поскольку для массивов в Ruby характерна динамическая длина, класс можно модифицировать для реализации объектов стека любой длины.

В силу того, что синтаксис имени класса отличается от синтаксиса имени экземплярной переменной, язык Ruby обеспечивает более высокий уровень читабельности, чем тот уровень, который характерен для других языков.

Параметрический полиморфизм в АТД

Степень универсальности абстрактных типов данных можно повысить за счет применения родовых параметров, обеспечивающих настройку не только на значения, но и на типы, а также действия. В этом случае появляется возможность распространить область применения АТД на целое семейство типов, оперировать понятиями обобщенный тип и обобщенный алгоритм. Как мы уже обсуждали в главе 13, один из методов реализации полиморфизма предполагает объявление типа элемента в качестве параметра модуля. Это называется параметрическим полиморфизмом. В данном разделе речь пойдет об использовании параметрического полиморфизма на уровне АТД. Обсудим средства языков Ada, C++, Java и C# для создания родовых (обобщенных) абстрактных типов данных, которые применяют параметрическую настройку типов.

Родовые АТД в языке Ada

В качестве синтаксических контейнеров для родовых АТД в языке Ada предоставляются родовые пакеты. Механизм настройки этих пакетов такой же, как и для подпрограмм. Он уже обсуждался в главе 13.

В качестве примера приведем родовой АТД для стека:

```
generic
    type T is private;  -- формальный родовой параметр (любой нелимитир. тип)
    Size : Positive := 4;
        -- указан тип и значение по умолчанию
package Gen_ADT_Stack is
    type Stack is private;
    Stack_Error : exception;
    procedure Reset ( The : in out Stack );
    procedure Push ( The : in out Stack; Item : in T );
    procedure Pop ( The : in out Stack; Item : out T );
private
    type Stack_Index is new Integer range 0 .. Size;
    subtype Stack_Range is Stack_Index
                range 1 .. Stack_Index ( Size );
    type Stack_Array is array ( Stack_Range ) of T;
    type Stack is record
        Elements : Stack_Array;   -- массив элементов
        Tos : Stack_Index := 0;   -- указатель на вершину
    end record;
end Gen_ADT_Stack;
package body Gen_ADT_Stack is
    procedure Reset ( The : in out Stack ) is
    begin
        The.Tos := 0;
                -- установить указатель  в нуль (нет элементов)
```

```
    end Reset;
    procedure Push ( The : in out Stack; Item : in T ) is
    begin
        if The.Tos /= Stack_Index ( Size ) then
                        -- проверка заполнения
            The.Tos := The.Tos + 1;  -- указатель вверх
            The.Elements (The.Tos) := Item; -- запись элемента
        else
            raise Stack_Error;    -- ошибка переполнения
        end if;
    end Push;
procedure Pop ( The : in out Stack; Item : out T ) is
    begin
        if The.Tos > 0 then
            Item := The.Elements (The.Tos ); --счит.верхн.элемен.
            The.Tos := The.Tos - 1;  -- указатель вниз
else
            raise Stack_Error;
                -- ошибка удаления из пустого стека
        end if;
    end Pop;
end Gen_ADT_Stack;
```

Родовой АТД не может использоваться непосредственно. Вначале надо выполнить конкретизацию его пакета, настройку на конкретный тип и размер. Например, экземпляр данного родового стека, предназначенный для работы с элементами типа `Integer`, конкретизируется следующим объявлением:

```
with Gen_ADT_Stack;
package Stack_Int is new Gen_ADT_Stack ( T => Integer, Size => 8 );
```

Для построения иерархической библиотеки абстрактных типов данных могут использоваться не только обычные, но и родовые дочерние пакеты.

В языке Ada определены следующие правила:

1. Любой родитель может иметь родового потомка (потомок — это дочерний пакет).
2. Родовой родитель может иметь только родовых потомков.
3. Если родитель не является родовым модулем, то родовой потомок конкретизируется обычным способом (в любом месте, где он виден).
4. При родовом родителе конкретизация потомка выполняется на основе конкретизации родителя.

Объявление конкретизации имеет вид:

```
with < ИмяРодовогоРодителя.ИмяРодовогоПотомка >;
with < ИмяЭкземпляраРодителя>;
package < ИмяЭкземпляраПотомка > is new
    <ИмяЭкземпляраРодителя.ИмяРодовогоПотомка>
            (< ПараметрыНастройки>);
```

Следовательно, конкретизации потомка должна предшествовать конкретизация родителя.

Допустим, что родовой `Gen_ADT_Stack` должен быть расширен для включения двух дополнительных операций: `Top` (возвращает верхний элемент из стека, без удаления его из стека) и `Items` (возвращает текущее количество элементов в стеке). Этого можно добиться, создав дочерний пакет для родового пакета `Gen_ADT_Stack`.

Так как родительский класс является родовым, то и дочерний пакет тоже должен быть родовым. Его спецификация имеет вид:

```
generic
package Gen_ADT_Stack.Additions is
    function Top ( The : in Stack ) return T;
    function Items ( The : in Stack ) return Natural;
private
end Gen_ADT_Stack.Additions;
```

Возможна следующая реализация данного потомка:

```
package body Gen_ADT_Stack.Additions is
    function Top ( The : in Stack ) return T is
    begin
        return The.Elements ( The.Tos );
    end Top;
    function Items ( The : in Stack ) return Natural is
    begin
        return Natural ( The.Tos );
    end Items;
end Gen_ADT_Stack.Additions;
```

Родовой потомок считается объявленным внутри родового родителя (его спецификация как бы «присоединяется» к концу спецификации родителя).

Для конкретизации родителя и потомка используются следующие объявления:

```
with Gen_ADT_Stack;
package Stack_Pos is new Gen_ADT_Stack ( Positive, 50 );
with Stack_Pos, Gen_ADT_Stack.Additions;
package Stack_Pos.Pos_Additions is
            new Gen_ADT_Stack.Additions;
```

Для тестирования дочернего модуля можно применить программу:

```
with Ada.Text_IO, Ada.Integer_Text_IO,
            Stack_Pos, Stack_Pos.Pos_Additions;
use Ada.Text_IO, Ada.Integer_Text_IO,
            Stack_Pos, Stack_Pos.Pos_Additions;
procedure Main is
    Pos_Stack : Stack;
begin
    Push ( Pos_Stack, 10 );
    Push ( Pos_Stack, 20 );
    Put ("Элемент на вершине ");
    Put ( Top ( Pos_Stack )); New_Line;
    Put ("Количество элементов ");
    Put ( Items ( Pos_Stack ));
end Main;
```

Классы-шаблоны в языке C++

Язык C++ обеспечивает поддержку родовых абстрактных типов данных с помощью классов-шаблонов.

Покажем применение классов-шаблонов на примере стека. В этом случае родовым параметром шаблона может стать тип элементов в стеке:

```
# include <iostream.h> //** подключение станд. библиотеки ввода-вывода
template <typename T>  //** формальный родовой параметр T
```

```
class Stack  {
    private:    //**скрытые элементы
    T *stack_ptr;
    int   max_len;
    int   tos;
    public:     //** эти элементы видимы клиенту
        Stack ( )   { //** конструктор для 500 элементов в стеке
            stack_ptr = new T [500];
            max_len = 499;
            tos = -1
        }
        Stack (int size) { //** конструктор для переменного количества элементов
                           //** в стеке
            stack_ptr = new T [size];
            max_len = size - 1;
            tos = -1;
        }
        ~ Stack ( )  {delete [ ]  stack_ptr;}; //** деструктор
    void push (T number)  {
        if (tos  = =  max_len)
            cout << "Error in push – stack is full \n";
        else stack_ptr [++tos] = number;
    }
    void pop ( )  {
        if  (tos  = =  -1)
            cout << "Error in pop – stack is empty \n";
        else  tos --;
    }
    int top ( )  {return (stack_ptr [tos]);}
    int empty ( )  {return (tos  = =  -1);}
}
```

Как и в языке Ada, классы-шаблоны конкретизируются во время компиляции. Например, экземпляр класса-шаблона `Stack`, а также объект (экземпляр типизированного класса) создаются по следующему объявлению:

```
Stack<float> stk;
```

Следует учесть, что если экземпляр класса-шаблона для типа `float` уже создан, то нет нужды в создании типизированного класса. В этой ситуации создается только объект.

Подчеркнем, что в общем случае при обработке приведенного выше объявления создается и конкретный класс, и объект — экземпляр конкретного класса. В этом состоит отличие от языка Ada, где данный процесс разнесен на два этапа:

1. На первом этапе (отдельно) конкретизируется родовой АТД;
2. На втором этапе (на основе конкретного АТД) создается объект.

Еще раз обратимся к определению класса-шаблона `Stack`. Теперь в нем два конструктора, один из которых вызывается по умолчанию и приводит к созданию стека фиксированного размера. Второй конструктор позволяет задать нужный размер стека. Он вызывается при другой форме объявления:

```
Stack<float> stk(200);
```

Следовательно, приведенное определение класса-шаблона обеспечивает настройку по двум параметрам: типу элементов и количеству элементов стека.

Родовые классы в языке Java

В языке Java у родовых АТД в качестве родовых параметров должны быть классы. До появления в этом языке механизма родовых параметров роль обобщенных типов выполняли коллекции из библиотеки классов, такие как `LinkedList` и `ArrayList`. Все типы коллекций сохранялись в экземплярах класса `Object`, поэтому они могли содержать любые объекты (кроме примитивных объектов). Такие типы коллекций были способны нести в себе несколько типов (таких, которые являются классами). В связи с этим надо констатировать следующее:

- Любое извлечение объекта из коллекции должно сопровождаться явным приведением объекта к соответствующему типу.
- Возможно добавление в коллекцию любого элемента, даже если коллекция предназначена для объектов только класса `Integer`.
- Любые коллекции не могут содержать величины примитивных типов. Перед сохранением в `ArrayList` целую величину нужно положить в экземпляр класса `Integer`.

Например, рассмотрим следующий фрагмент:

```
//* Создать объект класса ArrayList
ArrayList theArray = new ArrayList();
//* Создать элемент
theArray.add(0, new Integer(72));
//* Получить первый объект
Integer theInt = (Integer) theArray.get(0);
```

В настоящее время классы коллекций, среди которых наиболее популярным считают `ArrayList`, могут стать родовыми классами. Такие классы конкретизируются вызовом конструктора `new` и передачей ему родового параметра, заключенного в угловые скобки. К примеру, класс `ArrayList` можно конкретизировать для сохранения объектов класса `Integer` и получить конкретный объект:

```
ArrayList <Integer> theArray = new ArrayList <Integer>();
```

Этот новый класс снимает две проблемы коллекций. В коллекцию `theArray` можно положить только объекты класса `Integer`. Более того, при извлечении объекта из коллекции не требуется его явное приведение.

Язык Java также включает интерфейсы для таких коллекций, как списки, очереди и множества. Пользователи могут определять свои собственные родовые классы. Например, мы можем записать:

```
public class TheClass<T> {
. . .
}
```

Данный класс можно конкретизировать и создать соответствующий объект следующим образом:

```
TheClass<String> theString;
```

У подобных пользовательских родовых классов есть определенные недостатки:

- они не могут хранить примитивные величины;
- элементы не могут индексироваться. Элементы должны добавляться к коллекциям методом `add`.

Давайте реализуем родовой стек на основе коллекции `ArrayList`:

```
import java.util.*;
public class NewStack<T> {
    private ArrayList<T> refStack;
    private int maxLen;
    public NewStack() { // A constructor
        refStack = new ArrayList<T> ();
        maxLen = 499;
    }
    public void push(T newValue) {
        if (refStack.size() == maxLen)
            System.out.println("Error in push–stack is full");
        else
            refStack.add(newValue);
    }
    public void pop() {
        if (refStack.isEmpty());
            System.out.println("Error in pop–stack is empty");
        else
            refStack.remove(refStack.size() - 1);
    }
    public T top() {
        return (refStack.get(refStack.size() - 1));
    }
    public boolean empty() {return (refStack.isEmpty());
}
```

Здесь последний элемент в `ArrayList` находится с помощью метода `size`, который возвращает количество элементов в структуре. Элементы удаляются из структуры при помощи метода `remove`.

Рассматриваемый класс может быть конкретизирован для типа `String` (с построением соответствующего объекта) следующим образом:

```
NewStack <String> theStack = new NewStack <String>();
```

Как упоминалось в главе 13, в языке поддерживаются подстановочные классы. Например, `Collection<?>` является подстановочным классом для всех классов коллекций. Это позволяет написать метод, который в качестве типа параметра может использовать любой тип коллекции.

Поскольку коллекция сама по себе является родовой, класс `Collection<?>` можно считать родовым для родового класса.

При работе с объектами подстановочного типа следует соблюдать предельную осторожность. Поскольку компоненты конкретного объекта этого типа имеют свой тип, объекты других типов уже не смогут добавляться в коллекцию. Приведем пример:

```
Collection<?> strColln = new ArrayList<String>();
```

Фактически, здесь запрещается добавление в коллекцию объектов, не имеющих типа `String`.

Достаточно просто определить родовой класс для работы только с ограниченным набором типов. Например, класс может объявлять переменную родового типа и вызывать с ее помощью метод `compareTo`. При попытке конкретизации такого класса для типа, который не имеет метода `compareTo`, должна фиксироваться ошибка. Для предотвращения возможности конкретизации родового

класса типом, не поддерживающим метод `compareTo`, нужно задать следующий родовой параметр:

```
<T extends Comparable>
```

В языке Java `Comparable` является интерфейсом, в котором объявлен метод `compareTo`. Если в определении класса будет записан этот родовой тип, то вводится запрет на конкретизацию любым типом, не реализующим `Comparable`. Использование зарезервированного слова `extends` в данном контексте выглядит, по меньшей мере, странно, очевидно разработчики намекали на ассоциацию с понятием подтипа.

Родовые классы в языке C#

Вначале в языке C# были определены лишь классы коллекций, которые сохраняли объекты любого класса. Это были классы `ArrayList`, `Stack` и `Queue`. Перечисленные классы принесли те же проблемы, что и классы коллекций языка Java. Родовые классы были добавлены в 2005 году. Пятью предопределенными родовыми коллекциями стали `Array`, `List`, `Stack`, `Queue` и `Dictionary` (класс `Dictionary` реализует хеши). Данные классы устранили проблемы, связанные со смешиванием типов в коллекциях и необходимостью явного приведения объектов при извлечении из коллекций. Пользователи тоже могут определять родовые классы. При этом в пользовательских родовых коллекциях разрешается индексация элементов (доступ через индексы). Чаще всего используются целочисленные индексы, но в качестве индексов можно применять и строки. К сожалению, подстановочных классов в языке C# нет.

Синтаксические контейнеры для множества типов

Большие программы, включающие в себя несколько тысяч строк, инициируют две практические проблемы. С точки зрения программиста, наличия коллекции подпрограмм и определений АТД явно недостаточно для того, чтобы организация программы была интеллектуально обозрима. Второй практической проблемой является перекомпиляция. Для относительно небольших программ повторная компиляция всей программы сравнительно проста. Но для больших программ стоимость перекомпиляция существенно возрастает. Возникает необходимость выделения тех сегментов программы, которые не подвергались изменениям. Очевидным решением обеих проблем является организация программы в виде фрагментов логически связанного программного кода и данных, каждый из которых может компилироваться без перекомпиляции оставшейся части программы. Будем называть такие фрагменты *физическими контейнерами*.

Контейнеры часто помещаются в библиотеки и становятся доступны для многократного использования. Люди создают большие программы по меньшей мере 60 лет, так что техника формирования контейнеров развивается уже довольно давно. В языках, где разрешены вложенные подпрограммы, программы могут быть организованы в виде вложения объявлений одних подпрограмм в другие подпрограммы более высокого уровня, которые их используют. Это может быть сделано

в языках Pascal, Ada, Fortran 95, Python и Ruby. Увы, но этот подход далек от идеала и не может считаться основным средством организации программ.

Контейнеры в языке C

Язык C не обеспечивает полной поддержки абстрактных типов данных, хотя АТД и контейнеры для множества типов могут моделироваться. В C набор связанных между собой функций и определений данных можно поместить в файл, который компилируется независимо от других файлов. Такой файл, рассматриваемый как библиотека, содержит реализацию своих сущностей. Интерфейс к такому файлу, состоящий из объявлений данных, типов и функций, помещается в отдельный файл, называемый заголовочным файлом. Представления типов могут быть скрыты путем объявления их как указателей на структуры. Полные определения подобных типов-структур требуются только в файле реализации. Данный подход имеет такие же недостатки, которые характерны для указателей как абстрактных типов данных в Ada-пакетах: естественные проблемы указателей и возможная путаница при присваиваниях и сравнениях указателей.

Заголовочный файл в исходной форме и откомпилированная версия файла реализации передаются клиентам. При использовании такой библиотеки заголовочный файл включается в текст программы клиента: используется команда препроцессору `#include`. В итоге ссылки на функции и данные в клиентской программе могут быть проверены. Команда `#include` явно регистрирует зависимость клиентской программы от библиотечного файла реализации. По сути, такой подход эффективно отделяет спецификацию от реализации контейнера. Хотя пользователь может схитрить и просто скопировать содержимое заголовочного файла в свою программу, не прибегая к помощи `#include`. Подобный фокус может сработать, поскольку `#include` просто копирует содержимое файла-операнда туда, где данная команда появилась. Однако возможны два последствия:

- теряется регистрация зависимости клиентской программы от библиотеки и ее заголовочного файла;
- автор библиотеки может изменить заголовочный файл и файл реализации, а клиент будет пытаться использовать новый файл реализации со старым, скопированным заголовочным файлом.

Допустим, что в старом заголовочном файле для переменной `a` был определен тип `int`, а код реализации был перекомпилирован с новым заголовочным файлом, где для `a` задан тип `float`. Возникает рассогласование, которое может обернуться крахом вычислений и ответственность за которое должен нести клиент.

Контейнеры в языке C++

Язык C++ предлагает два различных вида контейнеров:

- заголовочный файл и файл реализации, которые могут быть определены так же, как в языке C;
- заголовки и определения классов.

В силу сложности взаимодействия шаблонов C++ и раздельной компиляции, заголовочные файлы библиотек шаблонов C++ достаточно часто включают полные определения ресурсов, а не только объявления данных и протоколы подпрограмм; отчасти это объясняется использованием редактора связей C для программ на C++.

Когда для контейнеров используются обычные классы, файл заголовка класса несет в себе только прототипы элементов-функций, а определения функций находятся вне класса, в файле реализации. Это явно отделяет интерфейс от реализации.

Возможны ситуации, когда операции определены так, что используют два различных класса объектов. В этом случае операцию нельзя привязать к одному классу. Предположим, что имеются АТД для матриц и АТД для векторов. Требуется операция умножения вектора на матрицу. Реализация этой операции должна иметь доступ к элементам-данным как класса матрицы, так и класса вектора, но ни один из этих классов не является естественным «домом» для операторов ее тела. Более того, вне зависимости от варианта размещения, остается проблема доступа к элементам другого класса. В C++ для таких случаев предусмотрены функции категории «друг класса». Дружеская функция получает доступ к приватным элементам того класса, где она объявлена другом. Для операции перемножения матриц и векторов возможно следующее решение: операция объявляется вне классов для матрицы и вектора, но определяется как друг в каждом из этих классов. Следующий фрагмент иллюстрирует данное решение:

```
class Vector {//** Определение класса вектор
    friend Vector mult(const Matrix&, const Vector&);
    . . .
};
class Matrix { //** Определение класса матрица
    friend Vector mult(const Matrix&, const Vector&);
    . . .
};
//** Функция, использующая объект-матрицу и объект-вектор
Vector mult(const Matrix& m, const Vector& v) {
    . . .
}
```

Друзьями могут объявляться целые классы. В этом случае все приватные элементы класса становятся видимыми для всех элементов дружеского класса.

Пакеты языка Ada

Пакет в языке Ada — это основное средство для поддержки многократности использования программного текста и может рассматриваться как универсальный контейнер логически связанных программных элементов. При проектировании программ пакеты позволяют применить подход клиент-сервер. Пакет действует как сервер, который предоставляет своим клиентам (программам и другим пакетам) набор услуг.

Спецификации пакетов языка Ada могут включать любое количество объявлений данных и подпрограмм в их публичных и приватных секциях. Следовательно, они могут содержать интерфейсы для любого количества абстрактных типов данных,

а также любых других программных ресурсов. Таким образом, пакет является контейнером для размещения множества типов. В частности, в один пакет можно поместить определения матричного и векторного типов, а также операции для их совместной обработки, которые обсуждались в предыдущем подразделе. При этом никакие дружеские функции не требуются.

В данной главе уже приводились примеры использования пакетов для размещения обычных и родовых АТД, а также иллюстрировалось применение дочерних пакетов. В качестве дополнительного примера рассмотрим родовой пакет с четырьмя родовыми параметрами:

```
generic -- родовые параметры спецификации
    type Элемент is private; -- любой нелимитированный тип
    type Индекс is ( < > );  -- любой дискретный тип
    type Вектор is array ( Индекс range <> ) of Элемент;
            -- любой регулярный тип
    with function Сумма ( X, Y : Элемент ) return Элемент;
            -- формальная функция, применяемая
            -- к объектам типа Элемент
package Обработка_Векторов is  -- заголовок спецификации
    function Обработка ( A, B : Вектор ) return Вектор;
    function Сигма ( A : Вектор ) return Элемент;
end Обработка_Векторов;
package body Обработка_Векторов is -- тело родового пакета
    function Обработка ( A, B : Вектор ) return Вектор is
        C : Вектор ( A'Range );
    begin
        for i in A'Range loop
            -- использование формальной функции
            C ( i ) := Сумма ( A ( i ), B ( i ));
        end loop;
        return C;
    end Обработка;
function Сигма ( A : Вектор ) return Элемент is
        C : Элемент := A (A'First);
    begin
        for i in Индекс'Succ(A'First) .. A'Last loop
            -- использование формальной функции
            C := Сумма ( C, A ( i ));
        end loop;
        return C;
    end Сигма;
end Обработка_Векторов;
```

Используем конкретизацию этого пакета для вычисления дохода от двух филиалов фирмы (по каждому рабочему дню и за всю неделю). Необходимая программа имеет вид:

```
with Обработка_Векторов;
procedure Main is
    subtype Деньги is Integer range 0 .. 30_000;
    type День is ( Пн, Вт, Ср, Чт, Пт, Сб, Вс );
    type Доход is array ( День range <> ) of Деньги;
    package Обр_Цел_Вект is
        new Обработка_Векторов ( Деньги, День, Доход, "+");
    -- факт. параметры конкретизации родового пакета
use Обр_Цел_Вект;
```

продолжение ⇨

```
Филиал1 : Доход ( Вт .. Пт ) := ( 250, 350, 100, 200 );
Филиал2 : Доход ( Вт .. Пт ) := ( 100, 250, 350, 150 );
Фирма : Доход ( Вт .. Пт );
Общ_Прибыль : Деньги;
begin
    Фирма := Обработка ( Филиал1, Филиал2 );
        -- прибыль по дням : Вт .. Пт
    Общ_Прибыль := Сигма ( Фирма );   -- прибыль за все дни
end Main;
```

ПРИМЕЧАНИЕ

Фактические родовые параметры должны строго соответствовать спецификации формальных родовых параметров. Если между ними существует логическая взаимосвязь, то это тоже учитывается. В частности, для нашего примера проверяется соответствие имени фактической родовой функции типу, который задается как первый параметр при конкретизации

Сборки языка C#

Язык C# имеет контейнер по имени *сборка* (assembly), который по своим размерам значительно больше класса. Сборка используется во всех языках программирования системы .NET. Сборки создают компиляторы .NET. Приложение .NET состоит из одной или нескольких сборок. Сборка — это файл, который появляется в прикладной программе или в виде динамически связанной библиотеки (dynamic link library, `.dll`), или в виде выполняемого кода (executable, `.exe`). Сборка определяет модуль, который может разрабатываться отдельно. В состав сборки входят различные компоненты. Одним из главных компонентов сборки считается программный код, представляемый на промежуточном языке, который является результатом компиляции исходного текста программы. В системе .NET промежуточный язык носит название `Common Intermediate Language` (CIL). Он используется во всех языках .NET.

В силу универсальности своего представления, сборка может использоваться в любой архитектуре, устройстве или операционной системе. При выполнении CIL-код сборки оперативно компилируется в машинный код компьютера.

Кроме того, .NET сборка содержит метаданные, описывающие каждый класс, который ее определяет, а также все внутренние классы, которые сборка использует. Сборка также содержит список всех сборок, упоминаемых в ней, и номер версии сборки. В пространстве .NET сборка считается базовым элементом разработки программной системы. Сборки могут быть приватными, в этом случае они доступны только одному приложению, или публичными, то есть обслуживающими любое приложение. В языке C# имеется уровень доступа `internal`. Элемент класса с таким уровнем доступа виден всем классам в той сборке, где он появился.

В языке Java существует файловая структура, подобная сборке и носящая название `Java Archive` (JAR). Она также используется при разработке программных систем на Java. JAR создается с помощью специальной утилиты `jar`.

Пространства имен

Мы обсудили синтаксические контейнеры для логически связанных программных ресурсов — в частности, абстрактные типы данных. Цель этих контейнеров — обеспечить организацию программ в виде логических единиц для компиляции. Они позволяют перекомпилировать фрагменты программ независимо друг от друга. В данном разделе рассматривается другая разновидность контейнеров, которая необходима для разработки больших программ: пространства имен.

Большая программа обычно создается многими программистами, работающими независимо друг от друга, возможно даже в разных географических регионах. Такой стиль разработки требует наличия логических фрагментов программы, независимых друг от друга, но способных работать совместно. При этом возникает проблема именования. Как обеспечить, чтобы имена переменных, методов, классов одного независимого разработчика не пересекались с именами, которые уже используются другим разработчиком в другой части той же программной системы?

Библиотеки являются источником той же проблемы именования. Более двух десятилетий растет (и стремительно прогрессирует!) зависимость больших программных систем от программных библиотек. Практически все программное обеспечение требует использования больших и сложных программных библиотек, вдобавок к специальным прикладным библиотекам. Широкое использование множества библиотек потребовало новых механизмов для управления именами. Например, если разработчик добавляет новые имена в существующую библиотеку или создает новую библиотеку, он не должен задействовать такое имя, которое конфликтует с именем, уже определенным в клиентской прикладной программе или в некоторой другой библиотеке. Без некоторой помощи языкового процессора это практически невозможно, потому что автору библиотеки просто негде узнать, какие имена уже появились в клиентской программе или что за имена из других библиотек клиентская программа может использовать.

Пространства имен фиксируют такие наборы имен, которые не станут источниками конфликтов. Каждая библиотека может создать свое собственное пространство имен, предотвращающее конфликты именования с другими библиотеками или с клиентскими программами. Каждая логическая часть программной системы может создать пространство имен с теми же целями.

Пространства имен являются *логическими контейнерами*, которым нет нужды быть непрерывными. Различные фрагменты программ могут быть помещены в одно и тоже пространство имен, даже если они хранятся в разных местах. Обсудим пространства имен в языках C++, Java, Ada и Ruby.

Пространства имен в языке C++

Язык C++ содержит спецификацию `namespace`, которая помогает программам управлять глобальными пространствами имен. Каждую библиотеку можно поместить в свое собственное пространство имен и предварять какое-либо имя именем этого пространства (при использовании имени за пределами пространства). Допустим, что существует заголовочный файл АТД, который реализует стеки. Если

есть опасения, что некоторый другой библиотечный файл может заимствовать имя, которое уже используется в АТД стека, то файл, определяющий стек, можно поместить в собственное пространство имен. Достаточно разместить все объявления для стека в блоке пространства имен:

```
namespace theStackSpace {
    // Объявления стека
}
```

Файл реализации для АТД стека может ссылаться на имена, объявленные в заголовочном файле с помощью операции *разрешения области видимости* `::` (двойное двоеточие), как показано в следующем примере:

```
theStackSpace::tos
```

Файл реализации тоже можно поместить в блок пространства имен, идентичный тому, что использовался для заголовочного файла. В этом случае все имена из заголовочного файла будут доступны (видимы) непосредственно. Безусловно, такой прием упрощает обращение, но в ущерб читабельности, поскольку становится менее очевидно, где именно объявлено цитируемое имя.

Программа клиента может получить доступ к именам из пространства имен библиотечного заголовочного файла тремя способами. Во-первых, имена из библиотеки квалифицируются именем пространства имен. Например, для ссылки на переменную `tos` надо записать:

```
theStackSpace::tos
```

Это самый точный и ясный способ ссылки, скажем, из файла реализации.

Два других способа используют директиву `using`. Эта директива может применяться для квалификации отдельных имен из пространства имен, например, так:

```
using theStackSpace::tos;
```

Такое предложение делает видимым только имя `tos`, но не другие имена из пространства имен `theStackSpace`.

Директива `using` может использоваться и для квалификации всех имен из пространства, как в следующем примере:

```
using namespace theStackSpace;
```

Фрагмент, который включает эту директиву, получает непосредственный доступ к именам из пространства имен, в следующем стиле:

```
x = tos;
```

В общем случае, при создании пространств имен в C++ используют и более изощренные правила, мы обсудили лишь основы.

Пространства имен языка C# используют те же механизмы и средства.

Пакеты в языке Java

Язык Java предлагает специальную конструкцию для создания пространства имен — *пакет*. Пакет может содержать определения нескольких типов, причем типы в пакете находятся друг с другом в доверительных отношениях. Более конкретно: сущности, определенные в типах пакета, могут быть публичными, защищенными, или не иметь спецификатора доступа и тем самым быть видимыми из всех других типов пакета.

Говорят, что сущности без спецификатора доступа имеют пакетную область видимости (пакетный уровень доступа), поскольку они видимы в любом месте пакета. В языке Java нет явных объявлений друга, отсутствуют дружеские функции и классы в интерпретации языка C++.

Ресурсы файла помещаются в конкретный пакет с помощью объявления:

```
package pkgstk;
```

Объявление пакета должно быть записано в первой строке файла. Ресурсы файла без объявления пакета неявно размещаются в некотором пакете без имени.

Клиенты пакета ссылаются на его типы с помощью полностью квалифицированных имен. Например, если в пакете `pkgstk` находится объект `theStack`, то к нему обращаются по имени `pkgstk.theStack`. Кроме того, на переменную `tos` в объекте `theStack` можно сослаться по имени `pkgstk.theStack.tos`. В случае вложенных пакетов такой подход становится несколько громоздким, поэтому предлагается объявление импорта, например:

```
import pkgstk.theStack;
```

Теперь на объект `theStack` можно сослаться по короткому его собственному имени. Для обеспечения доступа ко всем именам в пакете в операторе импорта достаточно вместо имени типа (объекта) поставить звездочку. Например, если нужно импортировать все имена из пакета `pkgstk`, следует записать:

```
import pkgstk.*;
```

Импорт в Java является лишь механизмом сокращения имен. С помощью импорта нельзя получить доступ к скрытым внешним ресурсам. Для получения доступа к ресурсу нужно знать путь (переменная среды `CLASSPATH`) и имя пакета.

Оператор импорта в Java документирует зависимость пакета, в котором он появился, от пакета, чье имя указано в декларации. Когда импорт не используется, эта зависимость менее очевидна.

Пакеты могут быть вложены внутрь других пакетов. Например, `java.lang` служит обозначением пакета `lang`, вложенного внутрь более крупного пакета `java`. Пакет `java` выполняет единственную роль — служит «оболочкой», содержащей другие пакеты. Стандартная библиотека классов в языке Java определена как иерархия многих пакетов.

Пространства имен в языке Ada

Пакеты в языке Ada очень часто используются для структурирования библиотек, формирования их иерархии. Например, если `Child_Pack` является дочерним пакетом для пакета `Father_Pack`, то говорят о двухуровневой иерархии и обращаются к имени `Name` в `Child_Pack` по составному (квалифицированному) имени:

```
Father_Pack.Child_Pack.Name
```

Пакеты также определяют пространства имен. Видимость пакета внутри программной единицы обеспечивается предложением `with`. Например, следующее предложение делает доступным ресурсы и пространство имен пакета `Ada.Text_IO`:

```
with Ada.Text_IO;
```

Доступ к именам из пространства имен пакета `Ada.Text_IO` должен быть квалифицирован. Например, на процедуру `Put` из `Ada.Text_IO` ссылаются следующим образом:

```
Ada.Text_IO.Put
```

Для доступа к именам из пакета `Ada.Text_IO` без квалификации используют предложение `use`:

```
use Ada.Text_IO;
```

При наличии этого предложения на процедуру `Put` из `Ada.Text_IO` ссылаются по простому имени `Put`. Предложение `use` эквивалентно объявлению импорта в языке Java.

Модули в языке Ruby

Классы в Ruby служат пространствами имен в той же степени, что и классы в других объектно-ориентированных языках программирования. Кроме того, Ruby предлагает дополнительную конструкцию для пространства имени, называемую модулем. Обычно *модули* определяют наборы методов и констант. Таким образом, модули удобны для создания библиотек соответствующих методов и констант, чьи имена находятся в отдельных пространствах имен, что исключает конфликты именования с другими именами в программе, которая использует модуль. В отличие от классов модули не могут конкретизироваться, иметь потомков, а также определять переменные. Методы, определяемые в модулях, включают в свои имена имя модуля. Рассмотрим следующий фрагмент:

```
module TheStuff
    PI = 3.1415
    def TheStuff.the_method1(arg1)
    . . .
    end
    def TheStuff.the_method2(arg2)
    . . .
    end
end
```

Допустим, что модуль `TheStuff` сохраняется в своем собственном файле `TheStuffFile`. Для использования констант и методов из модуля `TheStuff` программа должна получить доступ к модулю. Для этого применяется метод `require`, параметром которого является имя файла, записываемое в форме строкового литерала:

```
require 'TheStuffFile'
. . .
TheStuff.the_method1(a)
. . .
```

Как видим, при обращении к методу указывается имя модуля.

Контрольные вопросы и упражнения

1. Сравните понятия «абстракция процесса» и «абстракция данных». Чем они схожи? В чем состоят их отличия?

2. Разработайте пример абстрактного типа для стека на языке Pascal, предполагая, что определение стека, операции с ним и код, который его использует, находятся в одной и той же программе.
3. Разработайте пример абстрактного типа данных для матрицы на знакомом вам языке, создав операции сложения, вычитания и умножения матриц.
4. Разработайте пример абстрактного типа данных для очереди на знакомом вам языке, создав операции добавления элемента в очередь, исключения элемента из очереди и проверки заполненности очереди.
5. Разработайте абстрактный тип данных для комплексных чисел на знакомом вам языке, создав операции сложения, вычитания, умножения, деления, выделения действительной и мнимой части комплексного числа и построения комплексного числа из двух констант с плавающей точкой, переменных или выражений.

Глава 16

Объектно-ориентированное и аспектно-ориентированное программирование

Эта глава начинается с введения в широкий круг понятий объектно-ориентированного подхода к программированию, иллюстрируемых с помощью языка визуального моделирования UML. Последовательно анализируются объекты и классы как основные строительные элементы объектно-ориентированных программ. Значительное внимание уделяется описанию отношений между объектами и классами. Затем обсуждаются возможности и природа наследования, его корректность, формы наследования. Описываются и исследуются самые разнообразные вопросы: синтаксис пересылки сообщений; статическое и динамическое связывание сообщения и метода; разновидности переопределения методов (замещение и уточнение). Далее рассматриваются особенности объектно-ориентированного программирования на различных языках. В заключительных разделах поясняются особенности и базовые понятия аспектно-ориентированного подхода, приводятся примеры аспектного программирования на языке AspectJ.

В середине восьмидесятых годов стало ясно, что прогресс в разработке программных приложений связан с повторным использованием накопленных программ. Инкапсуляция и контроль доступа в АТД делают их лучшим средством повторного использования. Проблема повторного использования АТД состоит в том, что в новой разработке обычно требуется модификация его свойств и возможностей.

Вторая проблема заключается в том, что АТД независимы, находятся на одном уровне, образуют плоскую систему классификации. В реальной же жизни наиболее распространена иерархическая классификация объектов, в которой имеются предки, родители и потомки. Отсюда вытекают трудности настройки АТД на реальную предметную область.

Основные понятия объектно-ориентированного подхода к программированию

Основная цель объектно-ориентированного программирования (ООП) — обеспечить повторное использование существующего ПО.

Основной инструмент решения этой задачи — АТД, к которому добавляется возможность наследования. При наследовании новый АТД получает от родителя его данные и операции. В результате наследования можно изменить некоторые операции, добавить новые элементы (данные и операции).

АТД с поддержкой наследования называются классами.

Классы

Класс — описание множества объектов, которые разделяют одинаковые атрибуты, операции, отношения и семантику (смысл). Любой объект — просто экземпляр класса.

Как показано на рис. 16.1, различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс) [29].

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Иными словами, интерфейс демонстрирует внешнему миру абстракцию класса, его внешний облик. Интерфейс в основном состоит из объявлений всех операций, применимых к экземплярам класса. Он может также включать объявления типов (классов), переменных, констант и исключений, необходимых для полноты данной абстракции. В конкретных языках программирования константы и переменные, формирующие структуру данных класса, имеют разные названия. Например, в языке Smalltalk используется термин переменная экземпляра (instance variable), в языке Delphi Pascal и Java — поле (field), а в языке C++ — элемент данных (data-member). Будем использовать эти названия как синонимы.

Рис. 16.1. Структура представления класса

Интерфейс может быть разделен на 4 части:

1) публичную (*public*), объявления которой доступны всем клиентам;
2) защищенную (*protected*), объявления которой доступны только самому классу, его наследникам и друзьям;
3) приватную (*private*), объявления которой доступны только самому классу и его друзьям;
4) пакетную (*package*), объявления которой доступны только классам, входящим в один и тот же пакет.

Другом класса называют класс (функцию), который (которая) имеет доступ ко всем частям этого класса (публичной, защищенной и приватной). Иными словами, от друга у класса нет секретов.

ПРИМЕЧАНИЕ

Уровни видимости интерфейса (реализации) в разных языках программирования имеют свои особенности. Все без исключения уровни поддерживает язык C++. В нем же реализован механизм друга по отношению к функциям и классам. По существу, этот механизм нарушает принцип инкапсуляции, поэтому применять его надо предельно осторожно. В языке Java поддерживаются публичный, защищенный и приватный уровни видимости; здесь нет явного разделения класса на интерфейс и реализацию (оба раздела включены в общее описание класса), но уровень видимости каждого элемента класса задается индивидуально. Друзья в языке Java не предусмотрены, но обеспечивается пакетный доступ (все классы, входящие в один и тот же пакет, имеют доступ друг к другу). В языке Ada разрешены лишь публичный и приватный уровни видимости. В языке Smalltalk все переменные экземпляров считаются приватными, а все методы остаются публичными.

Реализация класса описывает секреты поведения класса. Она включает реализации всех операций, определенных в интерфейсе класса.

Для наглядного графического представления классов очень удобно использовать унифицированный язык визуального моделирования UML (Unified Modeling Language). В нем класс изображается как прямоугольник с тремя секциями: в верхней секции содержится имя класса, в средней — перечень атрибутов (полей данных), а в нижней — перечень операций класса (рис. 16.2). В зависимости от ситуации секции атрибутов и операций классов могут не отображаться.

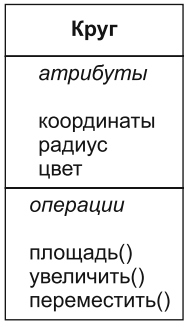


Рис. 16.2. Изображение класса в языке UML

Предусмотрено задание области действия атрибута (операции). Если атрибут (операция) подчеркивается — его областью действия является класс, в противном случае — областью действия является экземпляр (рис. 16.3).

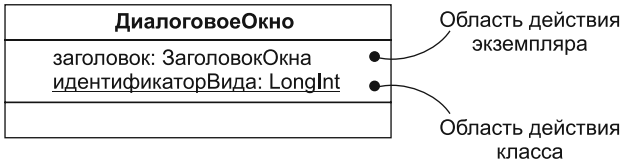


Рис. 16.3. Атрибуты уровней класса и экземпляра

Что это значит? Если областью действия атрибута является класс, то все его экземпляры (объекты) используют общее значение этого атрибута, в противном случае — у каждого экземпляра свое значение атрибута.

Общий синтаксис представления атрибута имеет вид:

```
Видимость Имя : Тип [Множественность] = НачальнЗначение {Свойства}
```

Рассмотрим видимость и свойства атрибутов.

В языке UML определены четыре уровня видимости.

Видимость	Пояснение
public	Любой клиент класса может использовать атрибут (операцию), обозначается символом +
package	Любой клиент класса, объявленный в том же пакете, может использовать атрибут (операцию), обозначается символом ~
protected	Любой наследник класса может использовать атрибут (операцию), обозначается символом #
private	Атрибут (операция) может использоваться только самим классом, обозначается символом –

ПРИМЕЧАНИЕ

Если видимость не указана, считают, что атрибут объявлен с публичной видимостью.

Свойств у атрибутов достаточно много, приведем лишь некоторые.

Свойство	Пояснение
readOnly	После инициализации объекта значение атрибута не изменяется (по умолчанию для атрибута установлено свойство changeable — нет ограничений на модификацию значения)
ordered	Для атрибутов с множественностью большей единицы; значения атрибута упорядочены
unique	Для атрибутов с множественностью большей единицы; наличие дубликатов значений запрещено

Примеры объявления атрибутов.

Атрибут	Пояснение
начало	Только имя
+ начало	Видимость и имя
начало : Координаты	Имя и тип
имяФамилия: String [0..1]	Имя, тип, множественность
левыйУгол : Координиаты=(0, 10)	Имя, тип, начальное значение
сумма : Integer [3]{readOnly, ordered}	Имя, тип, множественность и свойства

Общий синтаксис представления операции имеет вид:

```
Видимость Имя (Список Параметров): ВозвращаемыйТип {Свойства}
```

Примеры объявления операций.

Операция	Пояснение
записать	Только имя
+ записать	Видимость и имя
зарегистрировать(и: Имя, ф: Фамилия)	Имя и параметры
балансСчета ( ) : Integer	Имя и возвращаемый тип
нагревать ( ) {guarded}	Имя и свойство

В сигнатуре операции можно указать ноль или более параметров, форма представления параметра имеет следующий синтаксис:

```
Направление Имя : Тип = ЗначениеПоУмолчанию
```

Элемент `Направление` может принимать одно из следующих значений.

Направление	Пояснение
in	Входной параметр, не может модифицироваться
out	Выходной параметр, может модифицироваться для передачи информации в вызывающий объект
inout	Входной параметр, может модифицироваться

Примеры свойств операций.

Свойство операции	Пояснение
leaf	Конечная операция, операция не может быть полиморфной и не может переопределяться (в цепочке наследования)
isQuery	Выполнение операции не изменяет состояния объекта
sequential	В каждый момент времени в объект поступает только один вызов операций. Как следствие, в каждый момент времени выполняется только одна операция объекта. Другими словами, допустим только один поток вызовов (поток управления)
guarded	Допускается одновременное поступление в объект нескольких вызовов, но в каждый момент времени обрабатывается только один вызов охраняемой операции. Иначе говоря, параллельные потоки управления исполняются последовательно (за счет постановки вызовов в очередь)
concurrent	В объект поступает несколько потоков вызовов операций (из параллельных потоков управления). Разрешается параллельное (и множественное) выполнение операции. Подразумевается, что такие операции являются атомарными

Отношения между классами

Классы не существуют в изоляции. Напротив, с отдельной предметной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов программной системы.

Между различными классами могут устанавливаться разнообразные отношения. Их обозначения, принятые в языке UML, показаны на рис. 16.4.

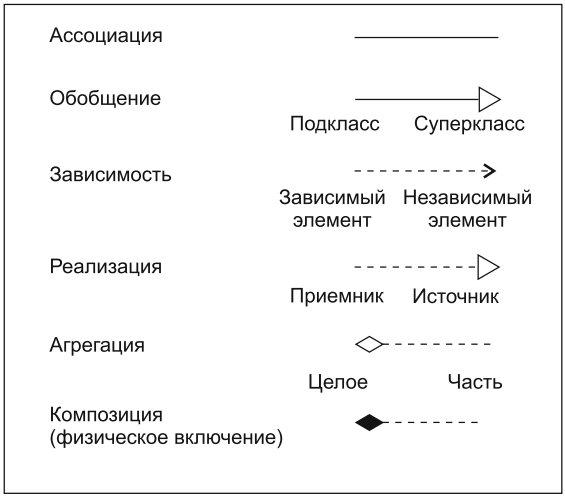


Рис. 16.4. Отношения между классами

Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы получаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек. *Ассоциации* отображают структурные отношения между экземплярами классов, то есть соединения между объектами.

Например, в системе обслуживания читателей имеются два ключевых класса — `Книга` и `Библиотека`. Класс `Книга` играет роль `элемента`, хранимого в библиотеке. Класс `Библиотека` играет роль `хранилища` для книг.

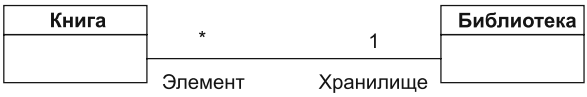


Рис. 16.5. Ассоциация между классами

Отношение ассоциации между классами изображено на рис. 16.5. Очевидно, что ассоциация предполагает двусторонние отношения:

- для данного экземпляра `Книги` выделяется экземпляр `Библиотеки`, обеспечивающий ее хранение;
- для данного экземпляра `Библиотеки` выделяются все хранимые `Книги`.

Здесь показана ассоциация один-ко-многим. Каждый экземпляр `Книги` имеет указатель на экземпляр `Библиотеки`. Каждый экземпляр `Библиотеки` имеет набор указателей на несколько экземпляров `Книги`.

Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. С помощью создания ассоциаций мы приходим к пониманию участников семантических связей, их ролей, мощности (количества элементов).

Ассоциация *один-ко-многим*, введенная в примере, означает, что для каждого экземпляра класса `Библиотека` есть 0 или более экземпляров класса `Книга`, а для каждого экземпляра класса `Книга` есть один экземпляр `Библиотеки`. Эту множественность обозначает *мощность ассоциации*. Мощность ассоциации бывает одного из трех типов:

- один-к-одному;
- один-ко-многим;
- многие-ко-многим.

Рис. 16.6. Ассоциации с различными типами мощности

Примеры ассоциаций с различными типами мощности приведены на рис. 16.6, они имеют следующий смысл:

- у европейской жены один муж, а у европейского мужа одна жена;
- у восточной жены один муж, а у восточного мужа сколько угодно жен;
- у заказа один клиент, а у клиента сколько угодно заказов;
- человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 16.7, роль ассоциации определяет, каким представляется класс на одном полюсе ассоциации для класса на противоположном полюсе ассоциации.

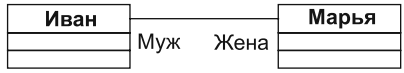


Рис. 16.7. Роли

Один и тот же класс в разных ассоциациях может играть разные роли.

Зависимость — это отношение, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его. Графически зависимость изображается как пунктирная стрелка, направленная на класс, от которого зависят. С помощью зависимости уточняют, какая абстракция является клиентом, а какая — поставщиком определенной услуги. Пунктирная стрелка зависимости направлена от клиента к поставщику.

Наиболее часто зависимости показывают, что один класс использует другой класс как аргумент в сигнатуре своей операции. Например, на рис. 16.8 показана зависимость класса `Заказ` от класса `Книга`, так как `Книга` используется в операциях `проверкаДоступности()`, `добавить()` и `удалить()` класса `Заказ`.

Рис. 16.8. Отношения зависимости

Ассоциации обозначают равноправные отношения между классами. *Агрегация* обозначает отношения классов в иерархии «целое/часть». Говорят, что агрегация образует «*part of*»-иерархию классов (и объектов). Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (атрибутам).

Агрегация не является понятием, уникальным для объектно-ориентированных систем. Например, любой императивный язык программирования, разрешающий структуры типа «запись», поддерживает агрегацию. И все же агрегация особенно полезна в объектно-ориентированном подходе к программированию.

Агрегация может обозначать, а может и не обозначать физическое включение части в целое. В нижней части рис. 16.9 приведен пример физического включения (композиции) классов-частей (`Slider`, `Header`, `Panel`) в класс-агрегат `Окно`. В этом случае говорят, что *части включены в агрегат по величине*.

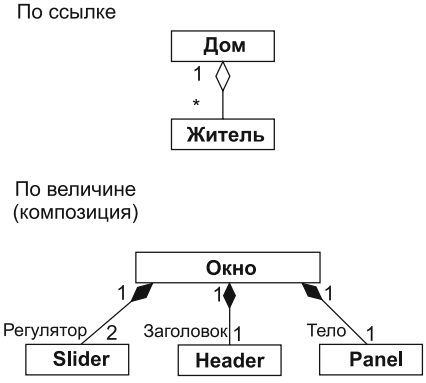


Рис. 16.9. Агрегация классов

В верхней части рис. 16.9 приведен пример нефизического включения класса-части `Житель` в класс-агрегат `Дом`. Очевидно, что жители достаточно часто находятся

в доме, но они не входят в него физически. В этом случае говорят, что *части включены в агрегат по ссылке*. Обратите внимание, что здесь указана множественность части в агрегате.

Реализация — это отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рис. 16.10 показано, что класс `Каталог` должен реализовать интерфейс `Обработчик каталога`, то есть `Обработчик каталога` рассматривается как источник, а `Каталог` — как приемник.

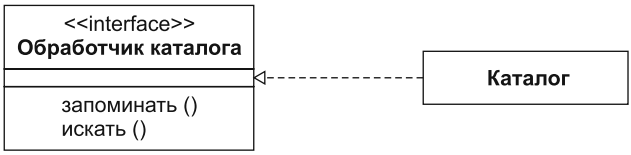


Рис. 16.10. Реализация интерфейса

Интерфейс `Обработчик каталога` позволяет клиентам взаимодействовать с объектами класса `Каталог` без знания той дисциплины доступа, которая здесь реализована (LIFO — последний вошел, первый вышел; FIFO — первый вошел, первый вышел и т. д.).

Обобщение — отношение между общей сущностью и специализированной разновидностью этой сущности. Наследование является наиболее популярной разновидностью отношения *обобщение-специализация*.

Наследование — это отношение, при котором один класс разделяет структуру и поведение, определенные в одном другом (простое наследование) или во многих других (множественное наследование) классах.

Между *n* классами наследование определяет иерархию «*является*» («*is a*»), при которой подкласс наследует от одного или нескольких более общих суперклассов. Говорят, что *подкласс* является специализацией его *суперкласса* (за счет дополнения или переопределения существующей структуры или поведения). Подкласс может иметь одного родителя (один суперкласс) или несколько родителей (несколько суперклассов). Во втором случае говорят о множественном наследовании.

Отношение наследования показывается с помощью стрелки с полым треугольным наконечником. Стрелка проводится от класса-наследника к классу-родителю. Наследника могут называть потомком, производным классом или дочерним классом, а родителя — предком, надклассом или базовым классом. Если у наследника один родитель, то это называют единичным наследованием. Например, на рис. 16.11 приведены классы-наследники `Круг` и `Квадрат`, родителем которых является класс `Точка`. `Круг` и `Квадрат` в результате наследования получают от `Точки` все ее атрибуты и операции.

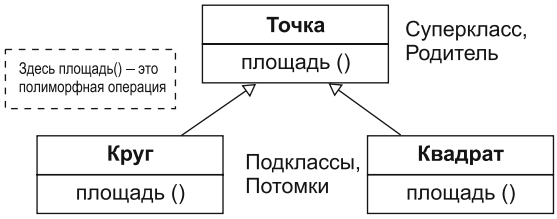


Рис. 16.11. Пример единичного наследования

При множественном наследовании у потомка несколько родителей (рис. 16.12).

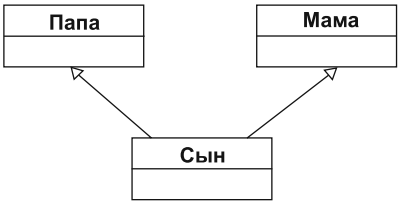


Рис. 16.12. Пример множественного наследования

И опять, следует говорить о том, что класс Сын унаследовал все папины и мамины атрибуты и операции. Теперь он является их наследником.

Рис. 16.13. Другой пример множественного наследования

Как показано на рис. 16.13, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу тоже достаются в наследство все атрибуты и операции двух классов-родителей. Правда, союз летающего предмета и хранилища вещей не столь естественен, как союз папы с мамой.

Множественное наследование достаточно сложно и коварно, имеет много «подводных камней». Например, подкласс Яблочный_Пирог не следует производить от суперклассов Пирог и Яблоко. Это типичное неправильное использование множественного наследования: потомок наследует все атрибуты от его родителя, хотя обычно не все атрибуты применимы к потомку. Очевидно, что Яблочный_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется *ромбовидная решетка наследования* (рис. 16.14).

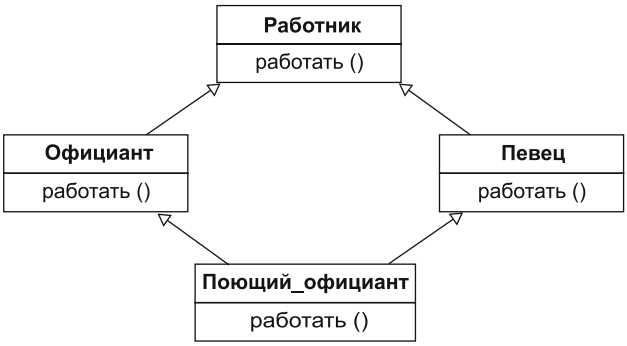


Рис. 16.14. Ромбовидная решетка наследования

Полагаем, что в подклассах `Официант` и `Певец` операция `работать()` суперкласса `Работник` переопределена в соответствии с обязанностью подкласса (работа официанта состоит в обслуживании едой, а певца — в пении). Возникает вопрос: какую версию операции `работать()` унаследует `Поющий_официант`? А что делать с атрибутами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий атрибута или только одну?

Все эти проблемы увеличивают сложность реализации, приводят к введению многочисленных правил для обработки особых случаев.

Деревья наследования классов

При использовании отношений наследования строится иерархия классов. Некоторые классы в этой иерархии могут быть абстрактными. *Абстрактным* называют класс, который не может иметь экземпляров. Имена абстрактных классов записываются курсивом. Например, на рис. 16.15 показаны абстрактные классы *Млекопитающие*, *Собаки*, *Кошки*.

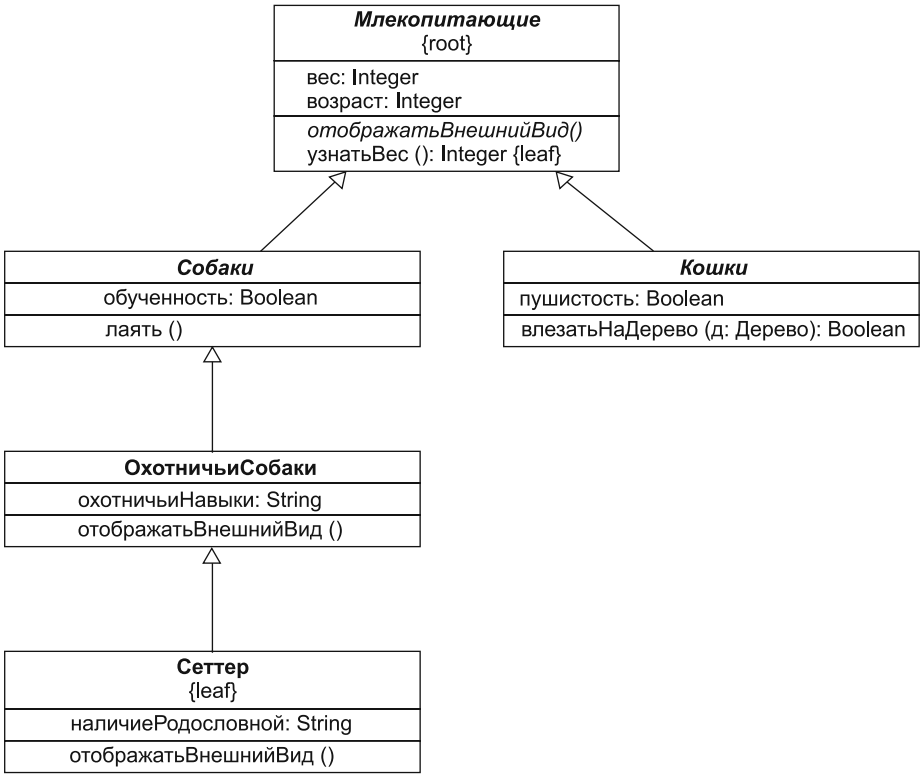


Рис. 16.15. Абстрактность и полиморфизм

Кроме того, здесь имеются конкретные классы `ОхотничьиСобаки`, `Сеттер`, каждый из которых может иметь экземпляры.

Обычно класс наследует какие-то характеристики класса — родителя и передает свои характеристики классу-потомку. Иногда требуется определить *конечный* класс, который не может иметь детей. Такие классы помечаются теговой величиной (характеристикой) `leaf`, записываемой за именем класса. Например, на рис. 16.15 показан конечный класс `Сеттер`.

Иногда полезно отметить *корневой* класс, который не может иметь родителей. Такой класс помечается теговой величиной (характеристикой) `root`, записываемой за именем класса. Например, на рис. 16.15 показан корневой класс *Млекопитающие*.

Аналогичные возможности имеют и операции. Обычно операция является полиморфной, это значит, что в различных точках иерархии можно определять операции с похожей сигнатурой. Такие операции из дочерних классов переопределяют поведение соответствующих операций из родительских классов. При обработке сообщения (в период выполнения) производится полиморфный выбор одной из операций иерархии в соответствии с типом объекта. Например, `отображатьВнешнийВид ( )` и `влезатьНаДерево ( )` — полиморфные операции. К тому же операция *Млекопитающие::отображатьВнешнийВид ( )* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, `Млекопитающие::узнатьВес ( )` — конечная операция, что отмечается характеристикой `leaf`. Это значит, что операция не полиморфна и не может перекрываться.

Объекты

Рассмотрим более пристально объекты — конкретные сущности, которые существуют во времени и пространстве (правда, в компьютерном времени и пространстве). В понятие программного объекта вкладывалась простая идея. Программные объекты должны моделировать физические объекты реального мира, поэтому они должны вобрать в себя черты и характеристики физических объектов.

Объект — это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Термины «экземпляр класса» и «объект» взаимозаменяемы. На рис. 16.16 приведен пример объекта по имени стул, имеющего определенный набор атрибутов и операций.

Индивидуальность — это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех атрибутов объекта и текущими значениями каждого из этих атрибутов (рис. 16.16).

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует то, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений. Поведение объекта — это его деятельность с точки зрения окружающей среды. Поведение объекта является функцией как его состояния, так и выполняемых им операций (`купить()`, `продать()`, `взвесить()`, `переместить()`, `покрасить()`). Говорят, что состояние объекта представляет суммарный результат его поведения.

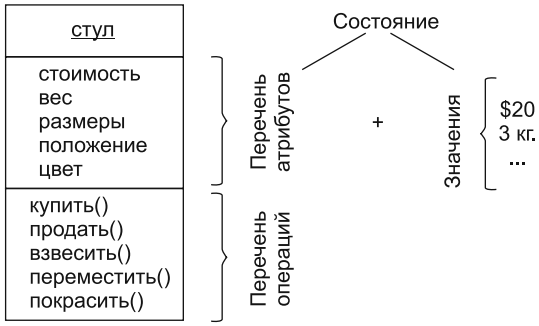


Рис. 16.16. Представление объекта с именем стул

Операция обозначает обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций клиента над объектом:

1) модификатор (изменяет состояние объекта);
2) селектор (дает доступ к состоянию, но не изменяет его);
3) итератор (доступ к содержанию объекта по частям, в строго определенном порядке);
4) конструктор (создает объект и инициализирует его состояние);
5) деструктор (разрушает объект и освобождает занимаемую им память).

Примеры операций приведены в табл. 16.1.

Таблица 16.1. Разновидности операций

Вид операции	Пример операции
Модификатор	пополнеть (кг)
Селектор	какойВес ( ) : integer
Итератор	показатьАссортиментТоваров ( ) : string
Конструктор	Робот (параметры) — создать
Деструктор	~Робот ( ) — уничтожить

В чисто объектно-ориентированных языках программирования операции могут объявляться только как методы — элементы классов, экземплярами которых являются объекты. Гибридные языки (C++, Ада) позволяют писать операции как свободные подпрограммы (вне классов). Соответствующие примеры показаны на рис. 16.17.

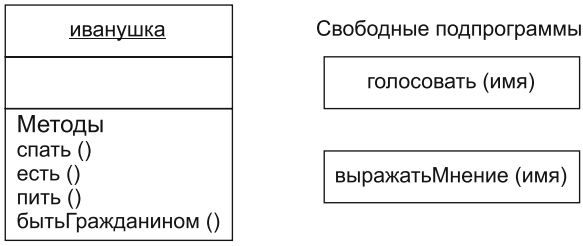


Рис. 16.17. Методы и свободные подпрограммы

С точки зрения внешней среды важное значение имеет такое понятие, как обязанности объекта. *Обязанности* означают обязательства объекта обеспечить определенное поведение. Обязанностями объекта являются все виды обслуживания, которые он предлагает клиентам. В мире объект играет определенные роли, выполняя свои обязанности.

Обозначение объекта показано на рис. 16.18.

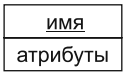


Рис. 16.18. Обозначение объекта

Имя объекта подчеркивается и указывается всегда, атрибуты указываются выборочно. И имя объекта, и имена атрибутов записываются с маленькой буквы.

Синтаксис представления имени имеет вид:

имяОбъекта : ИмяКласса

Примеры записи имени.

Имя	Пояснение
адам : Человек	Имя объекта и класса
: Пользователь	Только имя класса (анонимный объект)
мойКомпьютер	Только имя объекта (подразумевается, что имя класса известно)
агент :	Объект — сирота (подразумевается, что имя класса неизвестно)

Синтаксис представления атрибута имеет вид:

```
имя : Тип = Значение
```

Примеры записи атрибута.

Атрибут	Пояснение
номер:Телефон = “7350-420”	Имя, тип, значение
активен = True	Имя и значение

Объекты взаимодействуют друг с другом с помощью связей — каналов для передачи сообщений. Связь между парой объектов рассматривается как экземпляр ассоциации между их классами. Иными словами, связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами. Неявно все классы имеют ассоциацию сами с собой, следовательно, объект может послать сообщение самому себе.

Связь является путем для пересылки сообщения. Сообщение — это спецификация передачи данных между объектами в ожидании того, что будет обеспечена требуемая деятельность. Прием сообщения рассматривается как событие.

Результатом обработки сообщения обычно является действие.

Для записи сообщений в языке UML принят следующий синтаксис:

```
имяАтрибута = имяСообщения (Аргументы) : ВозвращаемоеЗначение
```

где `имяАтрибута` задает атрибут, куда помещается возвращаемое значение.

ПРИМЕЧАНИЕ

Прямо скажем, не очень прозрачный синтаксис. Наверное, `ВозвращаемоеЗначение` должно именовать какой-то атрибут объекта-адресата, в то время как `имяАтрибута` относится к объекту-источнику сообщения. Словом, здесь задействована как передающая, так и принимающая сторона.

Отношения между объектами

Отношения между парой объектов основываются на взаимной информации о разрешенных операциях и ожидаемом поведении. Выделяют два вида отношений между объектами: связи и агрегация.

Связь — это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

- объект-клиент вызывает операции объекта-поставщика;
- один объект перемещает данные к другому объекту.

Можно сказать, что связи являются рельсами между станциями-объектами, по которым ездят «трамвайчики сообщений».

Итак, взаимодействие объектов заключается в посылке друг другу сообщений. Сообщения имеют естественную «земную» интерпретацию — они указывают задания, которые должны выполнять объекты. Каждое сообщение именует операцию, которую объект-отправитель заказывает у объекта-получателя. После приема сообщения объект непременно начнет выполнять заказанную операцию. Чтобы обмен сообщениями состоялся, между объектами должны существовать связи — каналы для передачи сообщений. В языке UML связи рисуются простыми стрелками или линиями (если связь должна быть двунаправленной). Сообщения отображаются простыми стрелками, проставляемыми над линиями связей. Стрелки сообщений подписываются. В языках программирования принят следующий формат подписи сообщений:

```
<Имя объекта-приемника>.<Название операции>
```

В языке UML имя приемника не указывается: в этом нет необходимости, поскольку сообщение перемещается по связи, явно соединенной с получателем.

Приведем пример. При наличии классов `Мальчик` и `Девочка` можно объявить два объекта `петя` и `лена`:

```
петя : Мальчик;
лена : Девочка;
```

после чего объекты начинают обмениваться сообщениями (рис. 16.19).

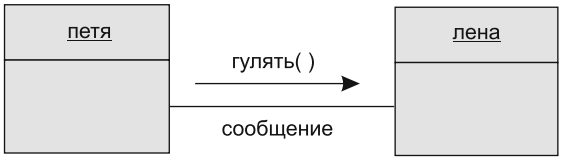


Рис. 16.19. Объект петя посылает сообщение объекту лена

На рис. 16.20 связи отображают возможные пути для передачи сообщений между четырьмя объектами. Сами сообщения показаны стрелками, отмечающими

их направления, и помечены именами вызываемых операций. На нашей диаграмме объект том просит объект мэри выполнить операцию `танцевать()`, а объект музыкальный_центр просит объект колонки исполнить операцию `звучать()`.

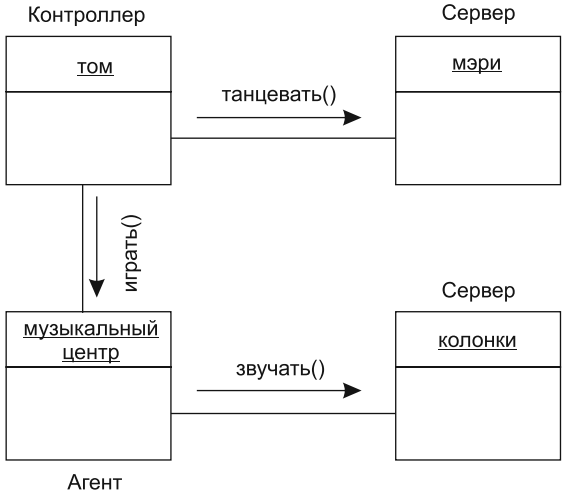


Рис. 16.20. Связи между объектами

Записывая имена `Мэри` и `Том` с маленькой буквы, мы вовсе не выказываем свое неуважение. Просто в программной среде имена объектов, их атрибутов и операций принято начинать со строчной буквы. Это одно из «правил приличия» проектирования и программирования.

Как участник связи объект может играть одну из трех ролей:

- *контроллер* — объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов;
- *сервер* — объект, который никогда не воздействует на другие объекты, он только используется другими объектами;
- *агент* — объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создается для выполнения работы от имени контроллера или другого агента.

На рис. 16.20 том — это контроллер, мэри, колонки — серверы, музыкальный центр — агент.

ПРИМЕЧАНИЕ

Полезно помнить о разной организации действий в физическом мире и в объектно-ориентированном программном мире. Поясним это на примере фразы «мама мыла раму». В физическом мире мама будет прикладывать действие к раме, изменяя ее состояние. В программном мире рама должна быть «самомоющейся». Объект мама может только послать сообщение «мыться» в объект раму, приняв которое рама начнет выполнять свою собственную операцию. Иными словами, в объектно-ориентированном подходе какому-либо объекту запрещается выполнять операцию, изменяющую значения атрибутов другого объекта. Обоснуйте, пожалуйста, смысл такого запрета.

Когда объект посылает сообщение в другой объект (делегируя некоторое действие получателю), объект-получатель, в свою очередь, может послать сообщение в третий объект и т. д. Так формируется поток сообщений — последовательность управления. Очевидно, что сообщения в последовательности должны быть пронумерованы. Номера записываются перед именами сообщений (через двоеточие), направления сообщений всегда указываются стрелками, размещаемыми возле линий связей.

Семантика и обозначения отношений агрегации между объектами полностью идентичны агрегациям между классами. Хотя нужно отметить, что на уровне объектов пометки мощности смысла не имеют и в агрегациях не применяются.

На этом мы завершаем краткий обзор базовых понятий объектно-ориентированного подхода и роли языка визуального моделирования UML в нем. За более подробным изложением этих вопросов следует обратиться к [29].

Возможности наследования и полиморфизм

Наследование является достаточно гибким механизмом, который позволяет наследнику не просто быть копией родителя, но и дополнять унаследованные черты своими собственными характеристиками. Наследник может иметь дополнительные атрибуты и операции. Наконец, он может изменить содержание унаследованной операции, переопределив ее.

Рассмотрим три типовых варианта применения наследования. Положим, у нас есть родительский класс с двумя атрибутами (элементами данных) `pa1` и `pa2` и двумя операциями (элементами-функциями) `op_a1()` и `op_a2()`. Этот родитель хочет, чтобы потомок был его точной копией. В этом случае потомок владеет теми же (родительскими) атрибутами и операциями, поэтому секции атрибутов и операций потомка пусты (рис. 16.21). Нет нужды показывать их содержимое — по умолчанию все знают, что там находится «родительское богатство».

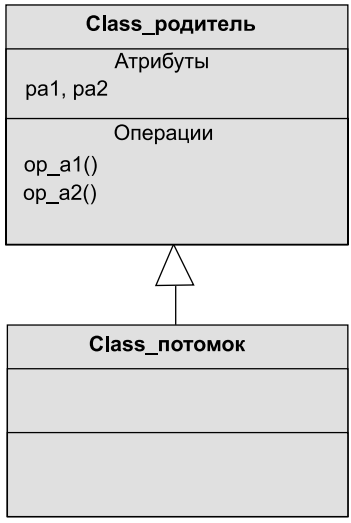


Рис. 16.21. Наследник является точной копией родителя

Однако второй потомок этого родителя-консерватора взбунтовался и реализовал идею: «я, папа, от твоего наследства не отказываюсь, но хочется чего-то новенького, дополнительного». В итоге наследник добавил себе два атрибута `pa3`, `pa4` и одну новую операцию `op_a3()` (рис. 16.22). Его секции атрибутов и операций не пусты — в них прописаны новые названия.

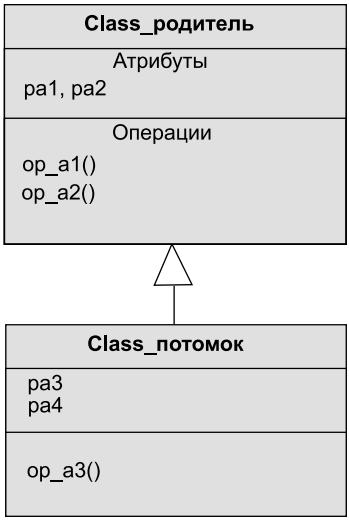


Рис. 16.22. Наследник наращивает количество атрибутов и операций

Наконец, третий потомок оказался самым требовательным: «я и нового хочу, и от старого отказываюсь (правда, лишь частично)». При реализации этого тезиса он разместил в себе новые атрибуты `pa5`, `pa6`, новую операцию `op_a4()` и переопределил родительскую операцию `op_a2()`, изменив ее содержание (рис. 16.23).

Переопределение родительских операций приводит к тому, что в дереве наследования классов появляются операции с одинаковым заголовком и разными телами — все вместе они образуют так называемую *полиморфную операцию*. Такая операция имеет много форм, то есть реализаций. На рис. 16.11 уже приводился пример полиморфной операции `площадь()`. Она появилась в силу переопределения классами-потомками `Круг` и `Квадрат` родительской операции, операции класса `Точка`. Возвращаясь к обсуждению полиморфизма в главе 13, следует сказать, что здесь вводится *полиморфизм подтипов*.

Для обращения к одной из форм полиморфной операции удобно использовать динамическое связывание. Оно создается на основе полиморфной переменной-указателя.

Полиморфная переменная-указатель привязывается к родительскому классу, после чего может адресовать любой из экземпляров классов-наследников. Например, мы объявляем указатель

```
Точка *фигура_укз; // полиморфная переменная-указатель на родительский класс
```

после чего с помощью оператора

```
фигура_укз -> площадь( );
```

получаем возможность обратиться к экземпляру как класса `Точка`, так и класса `Круг` или `Квадрат`.

Конечно, предварительно нужно занести в указатель адрес искомого экземпляра, а это можно сделать в любой момент периода выполнения программы.

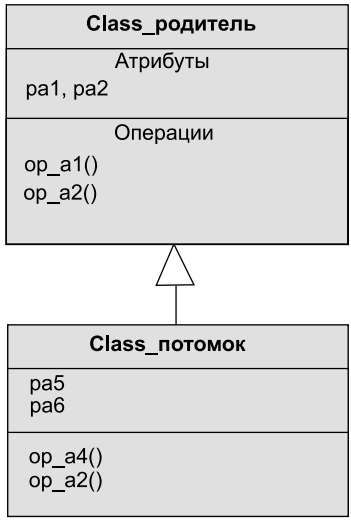


Рис. 16.23. Наследник наращивает количество атрибутов, операций и переопределяет родительскую операцию

Природа наследования

Представим себе класс `ВладелецКнижногоМагазина`. Мы вправе ожидать от объектов этого класса вполне определенного поведения не потому, что он владелец именно книжного магазина, а потому, что он владелец магазина. Например, он попросит вас оплатить заказ, а затем даст вам квитанцию. Эти действия не являются уникальными для владельца книжного магазина; они общие для владельцев обувных, продуктовых, галантерейных, винных и других магазинов. Следовательно, мы как бы связали определенное поведение с общей категорией «владельцы магазинов» `ВладелецМагазина`, и поскольку владельцы книжных магазинов являются частным случаем категории `ВладелецМагазина`, поведение для данного подкласса определяется автоматически.

В языках программирования наследование означает, что поведение и элементы данных, связанные с производным классом, всегда являются расширением характеристик, связанных с родительскими классами. Подкласс имеет все характеристики родительского класса и, кроме того, дополнительные характеристики. С другой стороны, поскольку производный класс считается более специализированной (или ограниченной) формой родительского класса, он также, в определенном смысле, будет сужением родительского класса. Это *диалектическое противоречие* между наследованием как расширением и наследованием как сужением является источником большой силы, присущей данному отношению, и в то же время вызывает некоторую путаницу.

Наследование всегда транзитивно, следовательно, класс может наследовать черты суперклассов, отстоящих от него на несколько уровней. Например, если `Сеттер` — это подкласс класса `ОхотничьиСобаки`, считающегося подклассом класса `Собаки`, а собаки являются подклассом класса `Млекопитающие`, то класс `Сеттер` наследует характеристики и охотничьих собак, и собак вообще, и всех млекопитающих `Млекопитающие`.

ПРИМЕЧАНИЕ

Отношение *R* называется транзитивным, если для любых *a*, *b*, *c* из *aRb* и *bRc* следует *aRc*.

Усложняющим моментом в механизме наследования является то обстоятельство, что подклассы могут переопределять поведение, унаследованное от родительского класса. Например, класс `Сеттер` переопределяет операцию `отображатьВнешний-Вид( )`, унаследованную от класса млекопитающих, поскольку внешний облик сеттеров очень специфичен. В данном разделе мы коротко затронем лишь некоторые аспекты переопределения.

Существует эмпирическое правило для проверки того, связаны ли два понятия отношением наследования. Оно называется проверкой «*is-a*» («*является*»). Тест «*является*» говорит, что если понятие `A` должно быть связано наследованием с понятием `B`, попробуйте произнести «`A` *является* `B`» (по-английски: «A(n) `A` *is a*(n) `B`»). Если ваш слух эта фраза не коробит, тогда наследование, скорее всего, уместно в данной ситуации. К примеру, вполне разумны следующие высказывания:

- `Птица` является `Животным` (A `Bird` is an `Animal`).
- `Кот` является `Млекопитающим` (A `Cat` is a `Mammal`).
- `Яблочный Пирог` является `Пирогом` (An `Apple Pie` is a `Pie`).
- `Текстовое Окно` является `Окном` (A `Text Window` is a `Window`).
- `Шар` является `Графическим Объектом` (A `Ball` is a `Graphical Object`).
- `Целый Массив` является `Массивом` (An `Integer Array` is an `Array`).

С другой стороны, следующие высказывания по той или иной причине кажутся странными и наследование, наверное, здесь не применимо:

- `Птица` является `Млекопитающим`.
- `Яблочный Пирог` является `Яблоком`.
- `Двигатель` является `Автомобилем`.
- `Текстовое Окно` является `Кнопкой`.
- `Целый Массив` является `Целой Величиной`.

Конечно, бывают ситуации, когда наследование применимо, даже если тест «*является*» не пройден. Тем не менее в большинстве случаев он считается хорошим индикатором целесообразности наследования.

Главными причинами применения наследования полагают:

- *Наследование как средство повторного использования программного кода.* Поскольку класс-потомок может наследовать поведение класса-родителя, исчезает

необходимость заново писать программный код для потомка. Это позволяет существенно сократить количество кода, требуемое для разработки новой идеи.

- *Наследование как средство повторного использования идеи (концепции, понятия).* Это происходит, когда класс-потомок переопределяет поведение, доставшееся ему от родителя. Хотя код реализации у потомка отличается от родительского, но идея какого-то метода у них общая. Следовательно, опять экономятся усилия, необходимые для новой разработки.

Иерархия наследования в различных языках

Объектно-ориентированные языки можно разделить на две категории:

1) языки, где требуется, чтобы каждый класс наследовал от существующего родительского класса;
2) языки, в которых такое требование отсутствует.

Примерами языков первой категории являются Java, C#, Smalltalk, Objective-C и Delphi Pascal, примерами второй — C++ и Apple (Object) Pascal.

Преимущество языков первой категории заключается в том, что все классы наследуют от существующего класса, который является единым корневым предком всех подклассов. В языках Smalltalk, Objective-C, Java, C# корневой класс называется `Object`, в Delphi Pascal он носит имя `TObject`. Любое поведение, обеспечиваемое корневым классом, наследуется всеми потомками. Таким способом гарантируется, что каждый объект обладает общим минимальным уровнем функциональности.

Недостаток языков первой категории проявляется в том, что единое дерево наследования объединяет все классы в большой «модуль» с высоким уровнем сцепления, иначе говоря, обуславливает существенную зависимость классов друг от друга. Если язык C++ разрешает несколько независимых иерархий наследования, то его программы избавлены от необходимости нести в себе большую библиотеку классов, из которой лишь немногие будут реально задействованы. С другой стороны, объекты в C++ не имеют гарантированной минимальной функциональности.

На организацию иерархии наследования отчасти влияет принятая в языке система типизации. В языках с динамической типизацией полезность объектов оценивается, прежде всего, перечнем воспринимаемых сообщений: чем больше сообщений может выполнить объект, тем удобнее его применять в вычислениях. В этих условиях совершенно естественно поощрять максимальное наращивание функциональности соответствующих классов через механизм наследования.

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков (LSP — Liskov Substitution Principle) считается индикатором, «лакмусовой бумажкой» правильности организации дочернего класса [29]. Авторская трактовка этого принципа достаточно замысловата: «Если каждому объекту `O1` типа `S` соответствует объект `O2` типа `T`, причем таким образом, что для всех программ `P`, определенных на основе `T`, поведение `P` не меняется при подстановке `O1` вместо `O2`, то `S` является подтипом `T`». Суть принципа в том, что

должна обеспечиваться корректность любой подстановки экземпляра дочернего класса вместо экземпляра родительского класса. Иными словами, дочерний класс обязан уметь делать все то, что умеет родительский класс.

LSP требует, чтобы любой производный класс соблюдал контракт между родительским классом и программными компонентами, которые его используют [76]. Контракт образуют:

1) предусловие, которое должно соблюдаться перед использованием компонентом родительского класса;
2) постусловие, которое должно соблюдаться после использования компонентом родительского класса.

При создании производных классов следует убедиться, что они соответствуют пред- и постусловиям контракта.

ПРИМЕЧАНИЕ

Программирование по контракту разработано Бертраном Мейером и реализовано им в языке Eiffel.

Эвристическим правилом для обнаружения класса, нарушающего LSP, является вопрос: удаляет ли этот класс какую-либо функциональность родительского класса?

Корректность наследования

Рассмотрим умозрительный эксперимент по идеальному, правильному применению механизма наследования. В этих условиях можно утверждать следующее:

- Экземпляр подкласса должен быть способен к обработке всех элементов данных родительского класса.
- Экземпляр подкласса должен реализовывать через наследование (если нет явного переопределения) все функциональные обязанности родительского класса. Это не противоречит возможности появления у потомка дополнительных обязанностей.
- Экземпляр подкласса должен имитировать поведение экземпляра родителя в такой степени, чтобы его было невозможно отличить от экземпляра родителя в сходных ситуациях.

Нетрудно заметить, что в данном эксперименте полученный подкласс удовлетворяет принципу подстановки Барбары Лисков и достоин называться подтипом.

При реальном применении наследования введенные утверждения не всегда верны. И предпосылки к этому имеются. Вспомним, например, упомянутое во втором утверждении «переопределение».

В силу этого условимся о следующей терминологии:

- Если подкласс удовлетворяет принципу подстановки, будем называть его подтипом.
- Если есть сомнения в том, что подкласс удовлетворяет принципу подстановки, будем называть его подклассом.

Все объектно-ориентированные языки поддерживают принцип подстановки, применяя дополнительные синтаксические правила для переопределения методов. Зачастую поддержка чрезвычайно прямолинейна: родительский класс просто интегрирует в себя содержимое дочернего класса. Наиболее отчетливым исключением является язык C++: в нем лишь указатели и ссылки полностью удовлетворяют принципу подстановки. Переменные, объявляемые как значения, принцип подстановки не обеспечивают. Разъяснения на этот счет приведем чуть позже.

Языки программирования со статической типизацией (например, C++) в большей степени опираются на принцип подстановки, чем языки с динамической типизацией (например, Smalltalk). Причина в том, что при статической типизации объекты проверяются через соответствующие им классы, а при динамической типизации — через их поведение. Например, статическая проверка полиморфной функции опирается на требование: все параметры должны быть подклассами «правильных» классов. Поскольку в языке с динамической типизацией параметры вовсе не имеют типа, требование меняется: параметр должен уметь выполнять задания, представленные в виде определенного набора сообщений. Скажем, при статической проверке функции требуется, чтобы параметр принадлежал к подклассу `Измеряемый`, а при динамической проверке — параметр должен «понимать» сообщения `меньшеЧем` и `равно`. Как мы поясняли, языки первой категории характеризуют объекты посредством их классов, а языки второй категории — посредством их поведения.

Переопределение и виртуальные методы

С точки зрения синтаксиса, при переопределении дочерний класс должен определить метод с тем же именем и сигнатурой типов, что и в родительском классе. Для сочетания переопределения с подстановкой необходимо обеспечить, чтобы объект, объявленный как экземпляр некоторого класса, дополнительно нес в себе черты объекта дочернего класса, тогда метод, инициируемый сообщением, можно найти в паре из этих классов. В подобных случаях требуется, чтобы был выполнен метод класса-потомка, а метод из класса-родителя был проигнорирован.

Таблица 16.2. Переопределение метода в языках C++ и C#

Язык	Пример переопределения

C++	`class ОхотничьиСобаки {`
	`public:`
	`virtual void отображатьВнешнийВид();`
	`};`
	`class Сеттер : public ОхотничьиСобаки {`
	`public:`
	`virtual void отображатьВнешнийВид(); // слово virtual здесь не обязательно`
	`};`

C#	`class ОхотничьиСобаки {`
	`public viritual void отображатьВнешнийВид() { ... }`
	`}`
	`class Сеттер: ОхотничьиСобаки {`
	`public override void отображатьВнешнийВид() { ... }`
	`}`

Во многих объектно-ориентированных языках (Smalltalk, Java) подобное поведение реализуется естественным образом, как только класс-потомок переопределит метод класса-родителя, сохранив сигнатуру типов в заголовке. С другой стороны, в некоторых языках потребуется, чтобы программист явно записал, что такая подстановка разрешается. Для индикации этой возможности во многих языках применяется ключевое слово `virtual`. В частности, в родительском классе C++ слово размещается в родительском классе (отмечая, что переопределение может произойти, но не указывая, что оно обязательно произойдет). В языке C# пометки делаются как в родительском, так и в дочернем классе, причем, как показано в табл. 16.2, используются разные слова.

Интерфейсы и абстрактные классы

Некоторые объектно-ориентированные языки, такие как Java, Ada, поддерживают концепцию, называемую интерфейсом. *Интерфейс* определяет протокол определенного поведения, но не обеспечивает его реализацию. Представим пример интерфейса, описывающего объекты, которые можно «*читать из*» и «*записывать в*» поток ввода-вывода:

```
public interface Transation {
    void writeOut (Stream s);
    void readFrom (Stream s);
};
```

Подобно классу, интерфейс определяет новый тип. Это значит, что переменные можно объявлять на основе имени интерфейса:

```
Transation buffer;
```

Класс может объявить, что он реализует протокол, определенный интерфейсом. Экземпляры класса могут назначаться переменным интерфейсного типа:

```
public class Memorization implements Transation {
    void writeOut (Stream s) {
    // ...
    };
    void readFrom (Stream s) {
    // ...
    };
};
buffer = new Memorization ();
```

Как и классы, интерфейсы могут наследоваться, причем разрешается наследование от множества родительских интерфейсов. Наследование классом-потомком от класса-родителя отличается от реализации классом интерфейса, но они очень похожи, поэтому в дальнейшем мы будем использовать термин «наследование» применительно к обоим действиям.

Отдельные объектно-ориентированные языки поддерживают идею, называемую абстрактным классом (или методом), которая соответствует чему-то среднему между классом и интерфейсом. В языках Java, Ada и C#, например, класс может определять себя и один или несколько методов с помощью ключевого слова `abstract`. У такого метода нет тела (реализации). Дочерний класс должен реализовать абстрактные методы, прежде чем на его основе смогут создаваться объекты. Таким образом,

абстрактные методы могут задавать идею поведения в родительском классе, но реализация поведения обеспечивается классом-потомком:

```
abstract class Window {
    ...
    abstract public void paint (); // рисовать содержимое окна
    ...
}
```

Весь класс может называться абстрактным, даже если в нем нет абстрактных методов. На основе абстрактного класса нельзя создать объект, но вполне допустимо рассматривать его как родительский класс в целях наследования.

В языке C++ идея абстрактного метода соответствует термину «чисто виртуальный метод», который определяется с помощью следующего оператора присваивания:

```
class Window {
    public:
    ...
    virtual void paint () = 0; // присваивание нуля делает метод чисто виртуальным
};
```

Класс может иметь как абстрактные (чисто виртуальные), так и не абстрактные методы. Класс, в котором все методы абстрактны, соответствует идее интерфейса из языка Java.

Формы наследования

Мир наследования в компьютерной среде даже разнообразней, чем мир наследования в среде человеческой. Очевидно, программисты активно используют опыт, накопленный человеческой цивилизацией. Попытаемся обсудить наиболее распространенные варианты [2, 18, 20, 26, 29, 34, 42, 76]. При этом будем обращать внимание на соблюдение принципа подстановки.

Наследование для специализации

Наиболее часто наследование применяют для специализации. В этом случае класс-потомок является специализированной формой класса-родителя, несущей в себе все родительские черты. Представим класс `Окно`, содержащий методы:

- `установить_размер()`;
- `изменить_положение()`;
- `отобразить()`;
- `изменить_размер()`;
- `свернуть()`;
- `развернуть()`.

Его наследник `ДиалоговоеОкно` предлагает дополнительные услуги: `ввести_данные()`, `вывести_данные()`. Поскольку подкласс `ДиалоговоеОкно` удовлетворяет всем характеристикам окна и предлагает специализированные операции, он удовлетворяет принципу подстановки и может быть назван подтипом окна.

Производный класс здесь считается более конкретным, частным или специализированным случаем родительского класса.

Возможны вариации этой формы наследования. В частности, считается корректным такое переопределение родительских методов, которое не меняет их суть, а просто добавляет детализирующие действия.

Наследование для реализации

Суть этой формы состоит в реализации некоторых идей, заложенных в родительском классе. Снова обратимся к классу `Окно`, но будем считать, что его методы `свернуть()` и `развернуть()` являются абстрактными, то есть не имеющими тел. Назовем эту версию `АбстрактноеОкно`. Подкласс `ЭкономичноеОкно` реализует абстрактные методы, сохраняя возможность предоставления клиенту всех остальных родительских методов (в унаследованном варианте). Он тоже получает квалификацию «подтипа окна».

В подобном наследовании класс-родитель лишь контурно описывает поведение, которое наполняется содержанием, то есть в полной мере реализуется классом-потомком.

По сути, речь идет об особом случае специализации, который связан с переводом родительских идей в практическую плоскость, их воплощением. То, что у родителя было зафиксировано как «сказка», в подклассе «становится былью».

Эти две формы наследования являются классическими, идеальными вариантами механизма наследования, целиком и полностью удовлетворяющими Барбару Лисков.

Наследование для конструирования

При таком наследовании родительский класс играет роль «детского конструктора», содержимое которого подкласс использует по собственному усмотрению, переименовывая методы и модифицируя списки их параметров. Словом, и внешний облик, и внутреннее содержание методов меняются до неузнаваемости. В итоге подобного использования, как правило, нарушается принцип подстановки и подкласс не может считаться подтипом родительского класса.

Например, путем наследования класса `ЭкономичноеОкно` конструируется класс `Самолет`, в котором родительский метод `изменить_положение()` превращается в метод `летать()`, а от методов `свернуть()` и `развернуть()` вообще отказываются. Или же можно вообразить конструирование подкласса `Очередь` на основе родительского класса `Стек`. В любом из этих случаев дочерний класс уже не является специализированной формой класса-родителя, так что нет никаких предпосылок для подстановки экземпляра потомка туда, где прописан экземпляр родителя.

В языках со статической типизацией наследование для конструирования объявлено «вне закона», так как оно напрямую нарушает принцип подстановки. Напротив, в языках с динамической типизацией его приветствуют как быстрый и простой способ создания новых абстракций. В частности, стандартная библиотека языка Smalltalk содержит большое количество подклассов, созданных конструированием.

Наследование для обобщения

Идея наследования для обобщения диаметрально противоположна наследованию для специализации. В этой форме подкласс модифицирует родительский класс для создания более общей сущности. Иными словами, вместо роли «подтипа» подкласс

претендует на роль «надтипа». Обобщение обычно применяется, если иерархия классов создается на основе существующих классов, которые нецелесообразно или просто нельзя изменять.

Будем считать, что имеется родительский класс `ДиалоговоеОкно`. Программистов вполне устраивает набор его методов, но появилась необходимость изменять цвет букв и цвет фона. Для этого надо добавить два атрибута `цвет_пера` и `цвет_фона` и переопределить родительский метод `отобразить()`, обеспечив в нем возможность работы с атрибутами цвета. В результате такой доработки появляется подкласс `ЦветноеДиалоговоеОкно`. Присвоить ему сертификат подтипа, увы, нельзя. Хотя, как мы увидим в дальнейшем, можно организовать такое переопределение метода, когда «невозможное станет возможным».

Отметим одну особенность: по большому счету, поведение подкласса, обеспечиваемое набором методов, сохранилось на уровне родительского, изменился лишь набор используемых атрибутов. Это типичная особенность наследования для обобщения.

Как правило, обобщений стараются избегать (если есть такая возможность). В нашем примере для этого достаточно «инвертировать ситуацию»: слегка изменить иерархию и содержание классов в библиотеке, подняв «цветовое решение» наверх.

Наследование для расширения

Если наследование для обобщения модифицирует существующее поведение родительского класса, то наследование для расширения добавляет совершенно новые черты поведения. Иначе говоря, наследование для обобщения обязано переопределить какие-то методы родителя, а функциональность подкласса жестко привязана к родительской функциональности. Напротив, расширение просто увеличивает родительский набор методов, и функциональность подкласса не так сильно привязана к существующим методам родителя.

Положим, что родителем является класс `Очередь`, реализующий дисциплину обслуживания `FIFO` и предоставляющий методы `вставить_с_хвоста()`, `удалить_с_головы()` и `пусто()`. Путем наследования для расширения создан дочерний класс `ДвухсторонняяОчередь`, в который добавлены два метода: `вставить_с_головы()` и `удалить_с_хвоста()`. Такие методы имеют смысл только для подкласса, но не для класса-родителя.

Поскольку методы родителя остаются нетронутыми и доступными, результаты наследования для расширения удовлетворяют принципу подстановки. Отсюда вывод: расширяющие подклассы всегда получают сертификат «подтипа».

Наследование для ограничения

Наследование для ограничения преследует цель: уменьшить функциональные возможности подкласса по сравнению с классом-родителем. Так же, как и в случае обобщения, необходимость в ограничении обычно возникает при создании подкласса на базе существующей иерархии классов, которые нецелесообразно или просто нельзя изменять.

Представим, что в библиотеке имеется родительский класс `ДвухсторонняяОчередь`, наследованием от которого нужно получить подкласс `Стек`. Как известно, стек реализует дисциплину обслуживания `LIFO` и должен предоставлять возможность

вставки и удаления элементов только с головы. Следовательно, в подклассе `Стек` нужно изъять родительские методы `вставить_с_хвоста()` и `удалить_с_хвоста()`. В этом и состоит ограничение.

Для воплощения ограничения используется одна из двух схем:

- закрыть доступ к запрещаемым методам;
- переопределить запрещаемые методы так, чтобы они лишь выдавали сообщения о невозможности использования.

В любом случае речь идет о переопределении родительского набора методов и, следовательно, о явном нарушении принципа подстановки. Результатом наследования для ограничения становится подкласс, не являющийся подтипом родительского класса. Таким образом, наследования для ограничения желательно избегать. Хотя на практике следовать подобному предостережению удается далеко не всегда.

Множественное наследование

Иногда требуется подкласс, наследующий характеристики двух или более родительских классов. Поющий официант, например, должен иметь черты как официанта, так и певца и, следовательно, способен демонстрировать двоякое поведение. Наследование от двух или более родительских классов известно как множественное наследование; оно способно породить странного монстра, поэтому в чистом виде во многих языках запрещено. Более безопасным считают множественное наследование на основе интерфейсов. В этом случае подкласс может наследовать характеристики одного родительского класса и реализовывать один или несколько интерфейсов. Например, на языке Java можно записать:

```
class ПоющийОфициант extends Официант implements Певец {
    ...
}
```

Здесь подразумевается, что `Официант` является классом, а `Певец` — это интерфейс.

Повышенная безопасность множественного наследования интерфейсов объясняется исчезновением двусмысленности имен, появляющихся в родителях:

- совпадающие имена методов с различными наборами параметров вполне различимы;
- полностью совпадающие сигнатуры родительских методов тоже не опасны, поскольку наследуется только спецификация, а не реализация.

Тем не менее язык C# запрещает наследование одноименных методов от двух родительских интерфейсов.

Выводы

Принцип подстановки предъявляет жесткие требования к корректности наследования. Далеко не все формы, применяемые на практике, выдерживают испытание в виде теста на подтип. К чему это приводит? Например, язык Java неявно предполагает, что все подклассы являются подтипами. Это означает, что экземпляр подкласса может быть назначен переменной, объявленной с типом родительского класса. Методы дочернего класса, имена которых дублируют родительские имена, могут переопределять унаследованное поведение. Мы уже видели, что допущение

«подкласс — это подтип» не всегда справедливо, поэтому подклассы без сертификата «подтип» становятся источниками возможных программных ошибок.

Тем не менее жизнь, то есть заказчик программного приложения, диктует свои законы. Достаточно часто программисты сознательно идут на нарушения. В этих ситуациях приходится формировать конкретные «правила игры». Проще всего запретить появление в программе экземпляров того класса, при наследовании от которого произошли нарушения, то есть «убить свидетелей-родителей». Насколько это целесообразно и этично? Другая возможность: создавать подпрограммы-обработчики, анализирующие возможность замещения родительского экземпляра «подозрительным» дочерним экземпляром и принимающие соответствующие решения. Появляется опасность «вместе с водой выплеснуть и ребенка», которым является мощный механизм наследования. Словом, на каждое «отклонение от закона» нужна оперативная реакция.

Вариации на тему наследования

Обсудим отдельные средства конкретных языков, ориентированные на поддержку наследования и переопределения.

Анонимные классы в языке Java

Иногда возникает ситуация, когда программисту требуется простой класс, обеспечивающий создание единственного объекта. Такой класс (объект) иногда называют *синглетоном*. Язык Java предлагает удобный механизм для создания подобного объекта при помощи безымянного класса, называемого здесь анонимным.

К созданию анонимного класса предъявляют следующие требования:

1. Может быть создан только один экземпляр анонимного класса.
2. Класс должен быть наследником родительского класса или интерфейса, в нем не предусматривается конструктор для инициализации.

Представим класс `ОбобщеннаяКнопка`, который применяется для создания графических кнопок. Для задания поведения кнопки следует сформировать новый подкласс, наследующий от класса `ОбобщеннаяКнопка` и переопределяющий ее метод `pressed`. Поскольку требуется только один такой объект, это можно сделать на основе анонимного класса.

Добавление в окно графического элемента выполняется с помощью метода `add`. Для размещения в окне новой кнопки достаточно сделать следующее:

```
Окно w = ...;
w.add (new ОбобщеннаяКнопка ("Quit") {public void pressed () { System.exit(0); } }
);
```

Обсудим параметр метода `add`. С помощью оператора `new` здесь задается создание экземпляра `Quit` анонимного подкласса, для которого родителем является класс `ОбобщеннаяКнопка`. Описание анонимного класса приведено в фигурных скобках. В описании указано определение метода, требуемого анонимному классу. В данном случае это метод, получаемый переопределением родительского метода `pressed`. Содержание метода записано во внутренних фигурных скобках.

Наследование и конструкторы

Конструктор, как вы помните, — это метод, который вызывается неявно при создании нового объекта и который гарантирует, что созданный объект правильно инициализирован. Наследование усложняет этот процесс, так как и родитель, и новый дочерний класс могут иметь свой собственный программный код для инициализации. Следовательно, в общем случае должен выполняться код в обоих классах.

В языке Java, C++ и других языках автоматически запускаются как родительский, так и дочерний конструкторы, их исполнение продолжается до тех пор, пока родительский конструктор не затребует дополнительных параметров. При возникновении таких требований дочерний конструктор явно обеспечивает их. В языке Java подобные действия задаются ключевым словом `super`:

```
class Child extends Parent {
    public Child (int y) {
    super (y + 7); // вызов родительского конструктора
    ...
    }
}
```

В языке C++ та же задача решается записью имени родительского класса в виде выражения инициализации:

```
class Child : public Parent {
public:
    Child (int y) : Parent(y + 7) { ... }
};
```

Аналогичным образом в языке Python метод инициализации вызывает родительскую функцию лишь тогда, когда программист явно запишет такой вызов:

```
class Child(Parent):
    def __init__ (self):
    # вначале инициализируется родитель
    Parent.__init__ (self)
    # после этого выполняется дочерняя инициализация
    …
```

Виртуальные деструкторы

В главе 15 пояснялось, что в языке C++ деструктор — это функция, которая автоматически вызывается для очистки памяти из-под объекта. Деструкторы используются для выполнения таких задач, которые гарантируют правильное удаление величины.

Достаточно часто деструктор освобождает динамическую память, в которой размещалась переменная.

Если ожидается подстановка и переопределение, очень важно, чтобы деструктор был объявлен как виртуальный. Невыполнение этого требования приводит к тому, что дочерние деструкторы не будут вызываться. Следующий пример иллюстрирует подобную ошибку:

```
class Parent {
public:
// Внимание! Здесь не объявлена виртуальность деструктора
~Parent () { cout << "в родителе\n"; }
};
```

продолжение ⇩

```
class Child : public Parent {
public:
    ~Child () { cout << "в потомке\n"; }
};
```

Если экземпляр дочернего класса был введен с помощью указателя на родительский класс, а затем должен быть уничтожен (например, оператором `delete`)

```
Parent *pointer = new Child();
delete pointer;
```

то будет вызван только родительский деструктор и на экране мы увидим:

```
в родителе
```

Если же родительский деструктор был объявлен как виртуальный, то будут выполняться как родительский, так и дочерний деструкторы. В C++ включение виртуального деструктора считается правильным решением, даже если он ничего не делает, поскольку у класса впоследствии могут появиться подклассы.

Сообщения и объекты

В этом разделе мы продолжим рассмотрение таких вопросов, как создание и инициализация новых объектов, а также их взаимодействие друг с другом путем пересылки сообщений.

Синтаксис пересылки сообщений

Термин «пересылка сообщения» обозначает динамический процесс обращения к объекту с требованием выполнить определенное действие. Пересылка сообщения отличается от обычного вызова подпрограммы:

- Сообщение всегда обращено к некоторому объекту, называемому получателем или адресатом.
- Действие, выполняемое в ответ на сообщение, не является неизменным и может варьироваться в зависимости от класса получателя. Различные объекты, принимая одно и то же сообщение, выполняют различные действия.

В выражении для пересылки сообщения имеются три четко различимые части: получатель (объект, которому посылается сообщение), селектор сообщения (текст, идентифицирующий конкретное сообщение) и список параметров, которые используются при реакции на сообщение.

Получатель	Селектор	Параметры
бармен	изготовить	(смесьНомер3, 0.25)

Как показано в табл. 16.3, синтаксис большинства языков использует точку для отделения получателя от селектора сообщения. В одних языках круглые скобки используются даже при пустом списке параметров, в других при отсутствии параметров скобки опускаются.

Таблица 16.3. Синтаксис посылки сообщений в различных языках

Языки	Выражение для пересылки сообщения

C++, C#, Java, Python, Ruby	barman.hello();
	barman.iClient(true);
	barman.toDrink(mixNumber3, 0.25);

Object Pascal, Delphi Pascal, Eiffel	barman.hello; barman. iClient (true); barman.toDrink (mixNumber3, 0.25);
Smalltalk	barman hello. barman iClient: true. barman toDrink: mixNumber3 at: 0.25
Objective-C	[barman hello]. [barman iClient: true]. [barman toDrink: mixNumber3 at: 0.25]
CLOS	(hello barman) (iClient barman true) (toDrink barman mixNumber3 0.25)

Языки Smalltalk и Objective-C применяют несколько другой синтаксис. В этих языках в качестве разделителя используется пробел. Сообщения без параметров просто записываются вслед за получателем. Сообщения с параметрами записываются с помощью нотации ключевых слов. Селектор сообщения разбивается на части, по одной части перед каждым параметром. За каждым ключевым словом следует двоеточие:

```
barman  toDrink:  mixNumber3 at:  0.25
```

В языке Smalltalk даже бинарные операции, такие как сложение, интерпретируются как посылка сообщения левой величине с правой величиной в качестве параметра

```
z <- x + y. " сообщение для x с заданием добавить к себе y и вернуть сумму "
```

В языке C++ тоже можно определять бинарные операции подобным образом. В языке Objective-C сообщения с синтаксисом, похожим на Smalltalk, заключаются в пару квадратных скобок и называются выражениями для передачи сообщений. В скобки заключается только само сообщение. В них не заключают переменную, которой присваивается результат сообщения:

```
int money = [barman getMoney ];
```

Язык CLOS следует традиционному синтаксису языка Lisp. Все выражения в Lisp записываются как списки, ограниченные парой круглых скобок. Операция является первым элементом списка, за которым следуют параметры. Получатель считается первым параметром.

Сообщения в языках со статической и динамической типизацией

В главе 8 уже обсуждались проблемы статической и динамической типизации языков. Вспомним, что в статически типизированных языках переменные сопостав-

ляются с соответствующими типами (обычно связывание задается объявлениями переменных), в то время как в динамически типизированных языках переменные рассматриваются просто как имена, а типы ассоциируются с их значениями. Языки Java, C++, C# и Pascal считаются статически типизированными языками, а Smalltalk, CLOS и Python являются динамически типизированными.

Язык Objective-C занимает среднее положение между этими двумя категориями. В Objective-C переменная может объявляться с фиксированным типом и тогда она считается статически типизированной. С другой стороны, переменная может декларироваться с помощью объектного типа `id`. Такая переменная может содержать любое объектное значение и в силу этого является динамически типизированной:

```
Barman oneBarmen; /* статически типизированная переменная */
id anotherBarmen; /* динамически типизированная переменная */
```

Применительно к передаче сообщений разница между статически типизированными языками и динамически типизированными языками очень важна:

- В статически типизированном языке тип получателя проверяется в период компиляции, это гарантирует, что получатель правильно воспримет адресованное ему сообщение.
- В динамически типизированном языке в период компиляции нет никакой возможности выполнить такую проверку.

Таким образом, в динамически типизированном языке, если получатель не сможет воспринять селектор сообщения, то будет сгенерирована ошибка периода выполнения. Такие ошибки в статически типизированном языке просто невозможны.

Доступ к получателю внутри метода

Как известно, сообщение всегда передается получателю. Тем не менее в большинстве объектно-ориентированных языков получатель не присутствует в списке параметров метода. Вместо этого получатель просто неявно вызывается из определения метода. В тех редких случаях, когда необходим доступ к получателю из тела метода, используют *псевдопеременную*. Псевдопеременная подобна обычной переменной, вот только она не объявляется и не может изменяться.

Псевдопеременная для получателя в языках Java и C++ получила имя `this`, в языке Eiffel — `Current`, а в Smalltalk, Objective-C, Object Pascal и многих других языках — `self`. Псевдопеременная используется для ссылки на экземпляр класса. К примеру, на языке Pascal может быть написан следующий метод:

```
function Student.study : progress;
begin
    if (self.mark = 4) or (self.mark = 5) then
        study:= Good
    else
    study:= Bad;
end
```

В большинстве языков псевдопеременную получателя разрешается опускать. Если запрашивается поле данных или вызывается метод без ссылки на получателя, по умолчанию считается, что предполагаемым адресатом сообщения является

псевдопеременная получателя. Рассмотрим метод перевода студента на следующий курс, в котором вызывается метод установки индикатора оплаты:

```
class Student {
    private boolean payment;
        ...
    public void nextCourse () { setUpPayment ( true ); }
        ...
}
```

Этот метод может быть переписан с указанием явного получателя:

```
class Student {
    private boolean payment;
        ...
    public void nextCourse () { this.setUpPayment ( true ); }
        ...
}
```

Псевдопеременную непременно нужно указывать, если метод хочет передать себя как параметр в другую функцию, что иллюстрирует следующий фрагмент на языке Java:

```
class ExitButton extends Button implements Action {
    public ExitButton () {
        ...
        // установить себя в качестве обработчика событий кнопки
        addAction(this);
    }
    ...
};
```

В языке Java рекомендуется использовать `this`, если параметры конструктора используются для инициализации элементов данных. Одно и то же имя может быть использовано как для параметра, так и элемента данных, поскольку явное `this` позволяет различать эти имена:

```
class Student {
    public Student (int course, int programme) {
    this.course = course; // this.course — это элемент данных
    this.programme = programme; // programme — это параметр
    this.payment = false;
    }
    ...
    private int course;
    private int programme;
    private boolean payment;
}
```

Некоторые объектно-ориентированные языки, такие как Python, CLOS, требуют, чтобы получатель явно объявлялся в описании метода. В языке Python, например, сообщение может быть записано с двумя параметрами:

```
theStudent.moveProgramme (1, 4)
```

но в соответствующем методе должны быть объявлены три параметра:

```
class Student:
    def moveProgramme (self, x, y):
    ...
```

В общем случае первый параметр может именовать все, что угодно, однако он используется для обозначения псевдопеременной получателя (`self` или `this`).

Создание объектов и конструкторы

Синтаксис, описывающий создание объектов в различных языках, показан в табл. 16.4.

Таблица 16.4. Синтаксис, используемый для создания объектов

Языки	Конструкция для создания объекта
C++	`Student *theStudent = new Student(1, 4);`
Java, C#	`Student theStudent = new Student (1, 4);`

Object Pascal	`var`
	`  theStudent : ^ Student;`
	`begin`
	`  new (theStudent);`
	`  ...`
	`end`

Objective-C	`theStudent = [Student new ];`
Python	`theStudent = Student (1, 4)`
Ruby	`theStudent = Student.new`
Smalltalk	`theStudent <- Student new`

Язык Python не применяет оператор `new` в явном виде, здесь объект создается при использовании имени класса в роли функции.

Повторим, что чаще всего для инициализации нового объекта применяется конструктор. Совмещение создания и инициализации дает определенные преимущества. Главное, что появляется гарантия того, что любому использованию объекта предшествует его корректная инициализация. Если создание и инициализация разделены, программист может просто забыть о вызове инициализирующей процедуры, зачастую с весьма печальными последствиями. Менее общей проблемой, хотя тоже неприятной, считается двукратный вызов процедуры инициализации. И этой проблемы можно избежать при использовании конструкторов.

В языках C++, Java и C# имя конструктора совпадает с именем класса, в котором он располагается. Кроме того, конструкторы не возвращают значения, возвращаемый тип в них не объявляется. Следующий пример иллюстрирует конструктор на языке Java:

```
class Student {
    public Student (int c, int p) {// конструктор в Java
    course = c;
    programme = p;
    payment = false;
    }
    ...
}
```

Когда память под объект выделяется в куче, посредством оператора `new`, параметры для конструктора записываются вслед за именем класса:

```
theStudent = new Student(1, 4);
```

Поля данных в языках Java и C#, которые инициализируются простыми значениями независимо от параметра конструктора, могут получать значения прямо в точке объявления, даже если они впоследствии будут изменены:

```
class ComplexNumber {
    public ComplexNumber (double r) { re = r; }
    …
    public double re = 0.0; // области инициализации данных
    public double im = 0.0; // нулями
}
```

Подобный синтаксис может использоваться и в C++, если элементы-данные объявлены с модификаторами `static` и/или `const`.

В языках C++ и Java, где допускается несколько определений функций с одинаковым именем и количеством параметров, для отличия функций друг от друга достаточно информации о типах и порядке следования их параметров. Эта возможность часто используется в конструкторах: разрешено создавать один конструктор без параметров, а другой — с тем или иным количеством параметров:

```
class Student {
public:
    Student ( ) // конструктор по умолчанию, без параметров
    { course = 1; programme = 4; payment = false; }
    Student (boolean is) // конструктор с одним параметром
    { course = 1; programme = 4; payment = is; }
    Student (int c, int p) // конструктор с двумя параметрами
    { course = c; programme = p; payment = false; }
};
```

Сочетание количества, типа и порядка параметров формирует сигнатуру (прототип) функции. Говорят, что смысл перегружаемого конструктора (или любой другой перегружаемой функции) определяется исследованием сигнатуры (прототипа) оператора вызова:

```
Student theStudent1; // вызывается конструктор по умолчанию
Student *theStudent2 = new Student;
Student theStudent3(true);
Student *theStudent4 = new Student(1, 1);
```

Обратите внимание, что в языке C++ при вызове конструктора по умолчанию скобки опускаются. Использование скобок в данном случае допускается, но имеет совершенно иной смысл:

```
Student theStudent5;   // создание нового объекта
Student theStudent6(); // опережающее определение функции
// с именем theStudent6, которая возвращает результат типа Student
```

С другой стороны, при использовании оператора `new` без аргументов скобки опускаются в C++, но не в языках Java или C#:

```
Student theStudent7 = new Student(); // Java
Student *theStudent8 = new Student;  // C++
```

Для определения начального значения элементов-данных конструкторы в C++ могут использовать несколько иной синтаксис. Двоеточие, следующее за именем в круглых скобках, рассматривается как начало инициализатора. При использовании синтаксиса инициализатора конструктор примет следующий вид:

```
Class Student {
public:
    Student (int c, int p)
    : course(c), programme (p), payment (false) { }
    …
};
```

Для простых значений (например, целых чисел) нет никакой разницы между инициализаторами и операторами присваивания в теле конструктора. Однако существуют такие формы инициализации, которые в C++ можно реализовать только при помощи инициализатора.

Имя конструктора в языке Objective-C отличается от имени класса. Определение конструктора начинается со знака «плюс». Такая функция называется фабричным методом. Фабричный метод использует оператор `new` для размещения объекта в памяти, а затем выполняет все действия, необходимые для инициализации объекта:

```
@ implementation Student
+ course: (int) c programme: (int) p {
    self = [Student new ];
    course = c;
    programme = p;
    return self;
}
@end
```

При вызове фабричных методов получателем считается класс, а не объект:

```
Student theStudent = [Student course: 1 programme: 1 ];
```

В языке Python все конструкторы имеют необычное имя `__init__`. При создании объекта неявно вызывается функция `init`, в качестве параметров ей передается имя объекта, а также другие параметры, используемые в выражении создания:

```
theStudent = Student (1, 4)
# вызывается Student.__ init__(theStudent, 1 4)
```

Связывание сообщения и метода

Поиск адресата сообщения обусловлен наличием в объектно-ориентированных программах полиморфных переменных, которые в разное время вычислений могут указывать на объекты разных типов (классов). Говорят, что такая переменная имеет динамический, меняющийся тип данных. В языках с динамической типизацией (Smalltalk, Objective-C) все переменные являются потенциально полиморфными, то есть любая переменная может содержать значение любого типа. В языках со статической типизацией полиморфные переменные возникают лишь при использовании указателей или ссылок. Полиморфизм существует в этих языках благодаря различию между декларированным (статическим) классом объекта и фактическим (динамическим) классом значения, которое содержится в адресуемом объекте. Это достигается через отношение наследования. Объект, адресуемый полиморфной переменной, может содержать значение объявленного типа или любого его подтипа. Возможны две точки зрения на тип полиморфной переменной:

- статическая (в соответствии с ее описанием в тексте программы);
- динамическая (в соответствии с ее текущим значением).

Первая точка зрения приводит к статическому связыванию сообщения с объектом-получателем, которое выполняется в период компиляции. Вторая точка зрения обосновывает динамическое связывание, которое откладывается до периода вычислений. Механизм статического связывания надежен, прост и эффективен; механизм динамического связывания значительно сложнее, но более гибок, чувствителен к текущей ситуации вычислительного процесса. Хотя накладные расходы на динамическое связывание могут быть весьма существенными.

Сложность динамического связывания определяется наличием в объектно-ориентированном пространстве целых семейств переопределенных методов. В этих условиях во время выполнения происходит постоянный поиск подходящего получателя сообщения. Рассмотрим, как решается задача динамического связывания в языках C++ и Java.

Связывание в языке C++

Связывание сообщений с методами в C++ организовано довольно сложно. Для обычных переменных (не указателей или ссылок) оно осуществляется статически. Но когда объекты задаются с помощью указателей или ссылок, применяется динамическое связывание. В последнем случае решение о категории связывания диктуется тем, описан ли соответствующий метод с помощью ключевого слова `virtual`. Если он объявлен именно так, то связывание сообщения основывается на динамическом классе, если нет — на статическом.

Допустим, что имеются следующие определения родительского и дочернего классов:

```
class Животное {
public:
    void speak() {
    printf("Не могу говорить!");
    }
};
class Сеттер : public Животное {
public:
    void speak() {
    printf("Гав, гав!");
    }
};
```

Объявим три переменные

```
Животное топ;
Сеттер элька;
Животное *бобик = new Сеттер;
```

и запишем программный фрагмент:

```
топ.speak();      // Не могу говорить!
элька.speak();    // Гав, гав!
бобик -> speak(); // Не могу говорить!
```

В качестве комментариев здесь указаны выводимые сообщения. Возникает вопрос: почему `бобик` не лает? Дело в том, что метод `speak()` в классе `Животное` не объявлен как виртуальный, поэтому компилятор считает, что у бобика статический тип `Животное`, а наши животные не лают.

Если же мы добавим в класс `Животное` спецификатор `virtual`

```
class Животное {
public:
    virtual void speak() {
    printf("Не могу говорить!");
    }
};
```

то по оператору

```
бобик -> speak();// Гав, гав!
```

бобик наконец-то залает.

Связывание в языке Java

В языке Java связывание сообщений и поиск подходящих методов значительно проще, чем в C++. В языке Java у любой переменной есть сведения о собственном динамическом типе данных. Считается, что все подклассы являются подтипами, и поэтому значение может быть присвоено объекту класса-родителя без явного преобразования. Допускается и обратное действие (присваивание экземпляру класса-потомка значения экземпляра класса-родителя), но уже с явным приведением типа. Для того чтобы определить допустимость присваивания, во время выполнения программы производится проверка и, если присваивание недопустимо, генерируется исключение:

```
Животное бобик;
  // ... пропущенный код
  // выполнить присваивание через приведение типа
Сеттер элька = (Сеттер) бобик;
```

О типах, находящихся на более высоких ступеньках иерархии классов, говорят как о *более широких*, или *менее конкретных*, по сравнению с теми, которые расположены ниже. Соответственно производные типы называют *более узкими*, или *более конкретными*, нежели их «прародители». Когда в выражении, предусматривающем использование объекта родительского класса, применяется объект производного класса, имеет место *преобразование с расширением типа*. Подобное действие, допустимость которого проверяется в период компиляции, приводит к тому, что объект дочернего типа интерпретируется как объект родительского класса. При этом программисту не нужно предпринимать никаких дополнительных усилий. Обратное действие, когда ссылка на объект родительского класса преобразуется в ссылку на объект дочернего класса, называют *преобразованием с сужением типа*. В этом случае нужно применить операцию *явного преобразования типа*.

Преобразование с расширением типа часто обозначают терминами *преобразование вверх* (upcasting), или *безопасное преобразование*, поскольку тип, расположенный на низкой ступени иерархии, приводится к классу более высокого уровня, а подобная операция заведомо допустима. Преобразование с сужением типа соответственно называют *преобразоваиием вниз* (downcasting) — здесь воздействию подвергается объект родительского типа, который должен быть интерпретирован как объект дочернего типа. Преобразование вниз считается *небезопасным преобразованием*, так как в общем случае его результат может быть неверен.

Когда компилятор, анализируя исходный текст программы, встречает выражение с явным преобразованием типов, он всегда анализирует корректность операции. Если его подозрения оправдываются, выдается донесение об ошибке. Если же компилятор не в состоянии оперативно оценить преобразование, он добавляет в код дополнительные команды для проверки в период выполнения.

Кроме того, динамический тип значения можно проверять с помощью операции `instanceOf`, возвращающей `true`, если выражение левой части совместимо с типом, название которого указано в правой части, и `false` — в противном случае. Используя `instanceof`, можно загодя убедиться в правомерности желаемого преобразования с сужением типа и избежать возникновения исключений:

```
if (бобик instanceOf Сеттер)
    ...
else
    ...
```

Для связывания сообщений с методами анализируют динамический тип данных получателя. В отличие от языка C++, здесь ключевые слова `virtual` отсутствуют. Интересно отметить, что в языке Java поля данных также могут переопределяться, но в этом случае доступ к переменной основывается на статическом типе данных, а не на динамическом. Обсудим определения двух классов:

```
class Parent {
    public void print() {
    println("Я отец!");
    }
}
    class Child extends Parent {
    public void print() {
    println("Я сын!");
    }
}
```

Для анализа их поведения используем следующий фрагмент:

```
class program {
    public static void main(String[] args) {
    Child c = new Child ();
    Parent p = c;
    c.print();  // печатает "Я сын"
    p.print();  // тоже печатает "Я сын"
    }
}
```

Как уже обсуждалось, помимо классов язык Java имеет синтаксическую конструкцию `interface`. Интерфейсы также обеспечивают иерархическую организацию, которая не зависит от иерархии классов. Используя ключевое слово `extends`, можно строить иерархию дочерних интерфейсов.

Интерфейсы позволяют определять переменные, которые способны воспринимать значения любых типов, реализующих интерфейсы. Считается, что интерфейс задает некоторый статический тип, в то время как каждый класс определяет динамический тип. Заметим, что связывание сообщений базируется на динамическом типе.

К примеру, построим следующую иерархию интерфейсов, задающую спецификации отдельных операций:

```
interface president {
    void planning();
    int controlManager();
}
interface manager extends president {
    void controlWorker();
}
interface worker extends  manager {
    int work(int);
}
```

В дальнейшем можно определить реализацию любого из интерфейсов:

```
class OurPresident implements president {
    // ...
}
```

Переопределение методов

Обычно функциональные возможности дочернего класса наращиваются за счет добавления дополнительных методов, имеющих новые имена и отличающихся от родительских как по спецификации, так и по реализации.

Тем не менее возможна и другая ситуация: дочерний класс вводит некоторый метод с тем же самым именем и параметрами, что и в родительском классе, но с иным содержанием. Такой метод дочернего класса переопределяет метод родительского класса.

В общем случае, когда сообщение посылается определенному объекту, поиск подходящего метода всегда начинается с проверки методов, определенных в классе данного объекта. Если ни один метод не найден, проверяются методы родительского класса. Если снова ни один метод не обнаружен, проверяется родительский класс родительского класса и т. д. В конечном счете находится подходящий метод, а при неблагоприятном исходе поиска формируется сообщение об ошибке.

Говорят, что метод в подклассе, имеющий то же имя и параметры, что и в суперклассе, переопределяет (override) метод суперкласса. Во время поиска метода, инициируемого приходом в объект сообщения, метод подкласса, конечно, будет найден раньше, чем одноименный метод в классе-родителе.

Метод дочернего класса может переопределять унаследованный метод путем замещения или уточнения. При замещении родительский код полностью перекрывается, то есть отбрасывается. Это значит, что код родительского метода никогда не задействуется при работе объектов дочернего класса. При уточнении вызов метода родительского класса является частью действий переопределяющего метода. Следовательно, родительское поведение целиком и полностью сохраняется и присоединяется к модифицированному поведению.

Замещение методов

Основная трудность при применении замещения методов в ходе наследования — решить вопрос с принципом подстановки, который настаивает на полном соблюде-

нии родительского поведения. Ведь новый вариант поведения может существенно отличаться от родительского.

Различные языки предлагают четыре варианта решения в споре «Барбара Лисков и принцип подстановки против замещения методов»:

1. Просто игнорировать такую постановку вопроса и обязать программиста самому заботиться о правильном поведении подклассов. По умолчанию подобное решение принято во всех обсуждаемых языках (C++, C#, Objective-C, Java и Smalltalk).
2. Бертран Мейер в своем языке Eiffel предлагает компромисс: присоединение к каждому методу логических утверждений, обеспечивающих проверку выполнения условий подстановки. Утверждения автоматически наследуются подклассами (в неизменной форме) и гарантируют допустимые рамки поведения. В настоящее время подобный подход постепенно распространяется и на язык C#.
3. Отделить понятие подкласса от понятия подтипа, как это отчасти сделано в языке Java за счет введения интерфейсов и механизма проверки. Подклассам выдать разрешение на использование замещения, освободив их от обязанности «быть подтипами».
4. Полностью запретить замещение, а использовать только уточнение. Эту возможность мы обсудим отдельно.

В языках используют различные подходы к указанию того, что некоторый метод переопределяется. Например, Smalltalk, Java и Objective-C вообще не требуют указания на переопределение. В C++ родительский класс должен иметь специальную пометку `virtual` о возможности переопределения. В языке Object Pascal это указание задается не в родительском, а в дочернем классе. В языке Delphi Pascal ключевое слово должно быть указано в обоих классах. Язык Ada просто рекомендует пометку `overriding`.

В языках C# и Delphi Pascal переопределение производится только в том случае, если родительский и дочерний методы оформлены специальным образом:

```
class Parent {   // пример C#
   public virtual int example (int a) { ... }
}
class Child : Parent {
   public override int example (int a) { ... }
```

Размещение указания в родительском классе несколько упрощает ситуацию, поскольку если нет возможности переопределить метод, то сообщение можно организовать через вызов процедуры, при котором динамический поиск в период вычислений не нужен. С другой стороны, отсутствие всякого рода уведомлений повышает гибкость языка, поскольку расширяет возможности применения механизма наследования (не надо обходить запреты).

Замещение методов в C++

Простое замещение применяется при выполнении двух условий:

1) параметры метода дочернего класса идентичны по типу и количеству параметрам метода родительского класса;
2) для описания метода в родительском классе используется модификатор `virtual`.

Рассмотрим два класса:

```
class Employee // некий сотрудник
{
  private:
    char name[ 40 ];      // имя сотрудника
    unsigned long number; // номер сотрудника
    int experience;       // стаж
  public:
    void getdata ( );
    void putdata ( );
    virtual bool isOutstanding();
};
class Professor : public Employee // профессор
{
  private:
    int pubs; // количество публикаций
  public:
    void getpubs ( );
    virtual bool isOutstanding();
};
```

В классе `Employee` метод `isOutstanding()` объявлен как виртуальная функция. В классе `Employee` этот метод возвращает `true`, если стаж работы больше 10 лет, в противном случае — `false`. Будем считать, что заслуги профессора оцениваются по другому параметру: количеству публикаций. Именно поэтому в классе `Professor` добавлен элемент данных `pubs`, метод для ввода количества публикаций `getpubs ( )`, а также замещен родительский метод `isOutstanding()`. Теперь метод `isOutstanding()` возвращает `true`, если количество публикаций больше 100.

Для компилятора языка C++ виртуальность метода `isOutstanding()` в классе `Employee` является прямым указанием на то, что связывание с этим методом *может* происходить динамически, в период выполнения программы. При описании в дочернем классе модификатор `virtual` необязателен. Как только метод объявляется виртуальным в родительском классе, он остается виртуальным и во всех подклассах. Однако для повышения читабельности этот модификатор обычно повторяется во всех дочерних классах.

Если модификатор `virtual` не задан, в экземпляре класса `Professor` метод по-прежнему будет замещать одноименный метод родительского класса. Однако процесс связывания метода и сообщения будет происходить статически, в период компиляции (исходя из объявленного статического типа получателя), а не во время вычислений, исходя из динамического типа получателя. Если ключевое слово `virtual` удалить из описания родительского метода `isOutstanding()`, то переменные, объявленные с типом `Professor`, будут выполнять метод из класса `Professor`, а переменные, объявленные с типом `Employee`, будут выполнять метод из этого класса независимо от действительного значения переменной.

Заметим, что конструкторы в C++ всегда используют семантику уточнения, а не замещения.

Замещение методов в Java

В языках Java, Smalltalk и Objective-C нет нужды уведомлять о переопределении метода. Достаточно того, что замещенный метод имеет то же самое имя, список параметров и тип возвращаемого значения, что и метод класса-родителя.

Интересные возможности в Java предоставляет ключевое слово `final`. Если оно записано в заголовке метода, то метод считается «листом» и переопределять его в подклассах нельзя. Если модификатор `final` записывается в заголовке определения класса, то у этого класса не может быть наследников. Компилятор языка Java обычно подставляет полный код реализации `final`-метода в точку, в которой он вызывается в программе.

Уточнение методов

Ранее мы описывали конфликт между переопределением и требованием принципа подстановки. Один из способов смягчения конфликта — изменение семантики переопределения. Вместо полного замещения родительского метода действия дочернего метода комбинируются с действиями родительского метода. Тем самым гарантируется, что действия класса-родителя будут выполняться во всех случаях, обеспечивая минимальный уровень функциональности. Желательность такого поведения наиболее очевидна при инициализации нового объекта. В этом случае целесообразно выполнить «родительские» действия по инициализации, а затем уже любые дополнительные действия, которые могут понадобиться для объекта класса-потомка.

В большинстве языков дочерний класс наследует полный доступ к данным и методам, поэтому добавить новую функциональность в метод проще всего простым копированием родительского кода. Но у этого подхода имеются очевидные минусы:

- снижается уровень совместного использования кода;
- уменьшается скрытность информации из родительского класса;
- понижается надежность, поскольку в ходе копирования могут появляться ошибки;
- исправление ошибок в родительском классе может не распространиться на производные классы.

Более привлекателен другой механизм внутри переопределяемого метода: прямой вызов метода-предшественника из родительского класса. В этом случае речь идет о повторном использовании кода переопределяемого родительского метода. Когда применяется такой механизм, говорят, что новый метод *уточняет* поведение родительского класса.

Уточнение в C++

В языке C++ вызов метода может записываться с помощью синтаксиса составного имени, при котором вместо используемого по умолчанию механизма поиска подходящего метода точно называется, из какого класса следует взять вызываемый метод. Строка составного имени имеет следующую структуру: имя класса, за ним два двоеточия, а затем имя метода. Напомним, что двумя двоеточиями обозначается операция разрешения области видимости. Применение составного имени выключает механизм поиска адресата и гарантирует, что метод будет вызываться из указанного класса.

Механизм составного имени используется для записи переопределения путем уточнения. Уточняющий дочерний метод явно вызывает родительский метод, за счет чего гарантируется исполнение обоих методов.

Снова обратимся к родительскому классу сотрудника и производному классу профессора:

```
class Employee // некий сотрудник
{
  private:
    char name[ 40 ];      // имя сотрудника
    unsigned long number; // номер сотрудника
    int experience;       // стаж
  public:
    virtual void getdata ( );
    virtual void putdata ( );
    virtual bool isOutstanding();
};
class Professor : public Employee // профессор
{
  private:
    int pubs; // количество публикаций
  public:
    virtual void getdata ( );
    virtual void putdata ( );
    virtual bool isOutstanding();
};
```

ПРИМЕЧАНИЕ

В целях упрощения примера конструкторы и деструкторы в классах Employee и Professor не показаны.

Здесь мы переопределили путем уточнения родительские методы `getdata ( )` и `putdata ( )`. Их модифицированное содержание в классе `Professor` принимает следующий вид:

```
    virtual void getdata ( )
      {
        employee::getdata ( );
        cout << "  Введите количество публикаций: "; cin >> pubs;
      }
    virtual void putdata ( )
      {
        employee::putdata ( );
        cout << "\n  Количество публикаций: " << pubs;
      }
```

Видим, что первичную работу по-прежнему выполняют родительские методы, за которыми следуют действия по обработке количества публикаций. Отметим, что теперь отпала необходимость в отдельном методе `getpubs ( )`.

Напомним, что конструктор отличается от других методов языка C++ тем, что в дочернем классе он всегда использует уточнение, а не замещение. Иначе говоря, дочерний конструктор всегда (и прежде всего) вызывает родительский конструктор.

Даже если в дочернем конструкторе нет явного вызова, все равно автоматически вызывается стандартный родительский конструктор без параметров. В противном случае следует явно указать параметры вызываемого родительского конструктора. Явный вызов требуемого родительского конструктора можно задать в заголовке

дочернего конструктора. Для этого нужно указать имя родительского класса и необходимые параметры. Например, конструктор класса `Professor`, который для вызова конструктора класса `Employee` использует два параметра, имеет вид:

```
Professor :: Professor (int a, char b, unsigned long c) : Employee (b, c)
{
  pubs = a;
}
```

Деструкторы в языке C++ задают противоположный порядок действий. Вначале выполняются действия дочернего деструктора, а затем следуют вызовы всех других деструкторов из суперклассов.

Уточнение в языке Java

Для разрешения переопределения метода путем уточнения в языках Java, Smalltalk и Objective-C используется псевдопеременная `super`. Если поиск соответствующего метода должен начинаться с родителя текущего класса, следует передать сообщение переменной `super`:

```
class Parent {
  private int x;
  public initialize()
  {
    x = 7;
  }
}
class Child extends Parent {
  private int y;
  public initialize()
  {
     y = 9;
     // выполнить метод родительского класса
    super.initialize();
  }
}
```

В языке Java текущий конструктор вызывает родительский конструктор с помощью ключевого слова `super`:

```
class subClass extends superClass
{
  subClass (int x, int y, int z)
  {
    // вызвать конструктор родительского класса
   super (x, y);
   // . . .
  }
}
```

ООП на языке C++

Объекты в C++ могут размещаться статически, динамически в стеке или размещаться в куче с использованием оператора `new`. Явное удаление в куче требует использования оператора `delete`.

Все классы C++ имеют минимум по одному конструктору. Конструктор неявно вызывается при создании объекта. Если какие-либо элементы-данные должны размещаться в куче, конструктор выполняет это размещение. Если в определении класса нет конструктора, то компилятор вызывает конструктор по умолчанию (из родительского класса). Если конструктор класса должен вызвать какой-то специальный конструктор из родительского класса (с конкретным набором параметров), то это задается в следующей форме:

```
Имя_подкласса(параметры): Имя_родителя(параметры) {
    . . .
}
```

Большинство классов включают деструктор, он неявно вызывается для удаления из кучи элементов-данных.

Единичное наследование

Форма записи класса-наследника имеет вид:

```
<класс_наследник> : <уровень_доступа>  <базов_класс>
    {<объявления элементов-данных,  элементов-функций>}
```

При уровне доступа `public` сохраняется уровень видимости элементов базового класса.

При уровне доступа `private` все элементы производного класса считаются приватными, то есть невидимыми для клиентов. Кроме того, приватные производные классы закрывают доступ ко всем элементам классов-предков со стороны всех классов-потомков.

Положим, что существует родительский класс:

```
class Parent_class  {
    private:
        int a;
        float x;
    protected: // разрешается доступ потомкам и друзьям
        int b;
        float y;
    public:
        int c;
        float z;
};
```

Будем считать, что у этого класса есть публичный и приватный наследники:

```
class Child_1: public Parent_class    {…};
class Child_2: private Parent_class  {…};
```

При приватном наследовании в экземплярах класса-наследника нет элементов, которые видят клиенты. Для восстановления видимости элемента применяется его повторное экспортирование. Оно отменяет скрытность элемента. Например:

```
class Child_3: private Parent_class {
    Parent_class ::  c;
    . . .
}
```

Теперь в экземплярах `Child_3` доступен элемент `c` (как если бы наследование было публичным). Двойное двоеточие — это операция разрешения области видимости. Она задает класс, где определен элемент.

Рассмотрим класс:

```
class Linked_list  {
    private:
        class Element  {
        public:
            Element *link;
            int  content;
        };
        Element *head;
    public:
        Linked_list  ( )  {head = 0};
        void  insert_at_head  (int);
        void  insert_at_tail  (int);
        int  delete_at_head  ( );
        int  empty  ( );
};
```

Вложенный в него класс `Element` определяет элемент связанного списка, включающий два компонента — поле целого типа и указатель на элемент. Этот класс размещен в приватной секции, что скрывает его от других классов. Элементы-данные в `Element` публичны, то есть видимы в объемлющем классе `Linked_list`. Если бы они были приватными, то для обеспечения видимости элементов-данных пришлось бы объявить класс `Element` другом класса `Linked_list`. Заметим, что у вложенных классов нет никаких специальных прав доступа к элементам объемлющего класса. Только статические элементы-данные объемлющего класса видимы методам вложенного класса.

Объемлющий класс `Linked_list` имеет один элемент-данные, это указатель на заголовок списка. Он содержит конструктор, обнуляющий значение указателя.

Четыре элемента-функции обеспечивают вставку элементов в список (с головы или хвоста), удаление элементов (с головы), проверку пустоты списка.

Определим классы стек и очередь, которые станут публичными потомками-наследниками класса связанный список:

```
class Stack : public Linked_list  {
    public:
        Stack  ( )  { };
        void push (int value)  {
            Linked_list :: insert_at_head (int value);
        };
        int pop  ( )  {
            return Linked_list :: delete_at_head  ( );
        };
};
class Queue : public Linked_list  {
    public:
        queue  ( )  { };
        void enqueue (int value)  {
            Linked_list :: insert_at_tail (int value);
        };
        int dequeue  ( )  {
```

продолжение ⇩

```
            return Linked_list  :: delete_at_head  ( );
        };
};
```

Заметим, что объекты как подкласса `Stack`, так и подкласса `Queue` имеют доступ к функции `empty ( )` из родительского класса `Linked_list` (так как это публичное наследование). Новые методы подклассов `push()`, `pop()`, `enqueue()`, `dequeue()` на самом деле лишь вызывают соответствующие методы родительского класса, которые и выполняют необходимую работу.

Оба подкласса содержат по конструктору (который ничего не делает). При создании объекта неявно вызывается конструктор подкласса. Далее вызывается конструктор родительского класса (в нашем примере именно он и выполняет необходимую инициализацию).

Однако здесь возникает серьезная проблема. Поскольку наследование публичное, объект класса `Stack` имеет доступ к операции `insert_at_tail ( )` родительского класса, а это разрушает целостность стека. Аналогично объекту класса `Queue` доступна операция `insert_at_head ( )`. Эти нежелательные вызовы вполне возможны, поскольку и класс `Stack`, и класс `Queue` являются подтипами типа `Linked_list`, а значит их экземпляры (в соответствии с принципом подстановки) могут использоваться вместо экземпляров `Linked_list`.

Если задуматься, то следует сказать правду: функциональное сходство стека и очереди со связанным списком весьма сомнительного свойства и не выдерживает теста «*является*», поскольку эти дети слабо похожи на своего родителя. Следовательно, нужно отказаться от родства и применить «запретное» наследование для конструирования, которое в языке C++ поддерживается механизмом приватного наследования. Заметим, что приватным подклассам придется скопировать метод `empty()`, поскольку он будет скрыт от их экземпляров. В итоге получим следующие подклассы:

```
class PrivStack : private Linked_list {
    public:
        PrivStack  ( )  { };
        void push (int value)  {
            Linked_list :: insert_at_head (int value);
        };
        int pop  ( )  {
            return single_linked_list :: delete_at_head  ( );
        };
    Linked_list :: empty  ( );
};
class PrivQueue : private Linked_list  {
    public:
        PrivQueue  ( )  { };
        void enqueue (int value)  {
            Linked_list :: insert_at_tail (int value);
        };
        int dequeue  ( )  {
            return Linked_list :: delete_at_head  ( );
        };
    Linked_list :: empty  ( );
};
```

Обратим внимание, что потребовался повторный экспорт операции `empty ( )`, поскольку она стала скрытой.

Две версии стека и очереди иллюстрируют различия между подтипами и подклассами, которые не являются подтипами. Связанный список может считаться обобщением стеков и очередей, но только на уровне реализации. С этой точки зрения вполне естественно наследование от класса связанного списка для определения классов стек и очередь. Тем не менее ни один из них не называется подтипом связанного списка, поскольку превращает публичные элементы родительского класса в приватные, запрещая клиентам соответствующий доступ.

Множественное наследование

Язык C++ поддерживает также множественное наследование. Синтаксис описания множественного наследования похож на синтаксис простого (единичного) наследования. Приведем пример:

```
class A {
};
class B {
};
class C : public A, public B {
};
```

Родительские классы подкласса `C` перечислены после двоеточия в строке описания класса и разделены запятыми.

В качестве иллюстрации множественного наследования рассмотрим менеджера, который когда-то учился и работал простым сотрудником. В классе `Student` содержатся сведения о школе или университете, которые закончил студент, и об уровне полученного им образования. Эти данные хранятся в строковом формате. Методы `getEdu ( )` и `putEdu ( )` позволяют нам ввести данные о студенте и просмотреть их. Предположим, что имеется информация об именах служащих всех категорий и их идентификационных номерах. Кроме того, в информации о менеджерах еще содержится название их должности и их взносы в гольф-клубы. Соответствующие определения классов имеют следующий вид:

```
class Student // сведения об образовании
{
  private:
    char school[ 70 ]; // название учебного заведения
    char degree[ 70 ]; // уровень образования
  public:
    void getEdu ( )
      {
        cout << "  Введите название учебного заведения: ";
        cin >> school;
        cout << "  Введите степень высшего образования\n";
        cout << "  (неполное высшее, бакалавр, магистр, доктор наук): ";
        cin >> degree;
      }
    void putEdu ( )
      {
        cout << "\n  Учебное заведение: " << school;
        cout << "\n  Степень: " << degree;
      }
};
```

продолжение ⇨

```
class Employee // некий сотрудник
{
  private:
    char name[ 40 ];      // имя сотрудника
    unsigned long number; // номер сотрудника
  public:
    void getData ( )
      {
        cout << "\n  Введите фамилию: "; cin >> name;
        cout << "  Введите номер: ";     cin >> number;
      }
    void putData ( )
      {
        cout << "\n  Фамилия: " << name;
        cout << "\n  Номер: " << number;
      }
};
class Manager : private Employee, private Student // менеджер
{
  private:
    char title[ 40 ];  // должность сотрудника
    double dues;       // сумма взносов в гольф-клуб
  public:
    void getData ( )
      {
        Employee::getData ( );
        cout << "  Введите должность: "; cin >> title;
        cout << "  Введите сумму взносов в гольф-клуб: "; cin >> dues;
        Student::getEdu ( );
      }
    void putData ( )
      {
        Employee::putData ( );
        cout << "\n  Должность: " << title;
        cout << "\n  Сумма взносов в гольф-клуб: " << dues;
        Student::putEdu ( );
      }
};
```

Динамическое связывание

До сих пор мы записывали статически связанные элементы-функции, то есть вызов одной из них статически связывался с определением функции.

В C++ для указания объектов класса-наследника может использоваться указатель или ссылочная переменная, которая имеет тип родительского класса. Такая переменная является полиморфной. Когда эта полиморфная переменная используется для вызова функции, определенной в одном из классов-наследников, вызов должен быть динамически связан с правильным определением функции.

Динамически связываемые элементы-функции должны объявляться как *виртуальные функции*. Их заголовки (в теле класса) начинаются со служебного слова `virtual`.

Положим, что есть родительский класс `Форма` и набор классов-наследников от формы (`Круг`, `Прямоугольник` и т. д.). Все эти фигуры отображаются с помо-

щью элемента-функции `display()`, которая должна быть уникальна для каждого подкласса (разновидности формы). Все версии `display()` должны определяться как виртуальные. Когда вызов `display()` выполняется с помощью указателя на родительский класс, этот вызов должен быть динамически (в период выполнения программы) связан с элементом-функцией нужного класса-потомка.

Мы должны создать следующие определения:

```
class Форма  {
    public:
        virtual void display() = 0;  // чисто виртуальная функция
        . . .
}
class Круг : public Форма  {
    public:
        virtual void display()  {…}
        . . .
}
class Прямоугольник : public Форма  {
    public:
        virtual void display()  {…}
        . . .
}
class Квадрат : public Прямоугольник {
    public:
        virtual void display()  {…}
        . . .
}
```

На основе этих определений допустимы следующие статические и динамические вызовы:

```
Квадрат *square new Квадрат; // создание объекта в куче
Прямоугольник rectangle;     // создание объекта в стеке
Форма *ptr_shape;    // создание указателя на экземпляры класса Форма
ptr_shape = square;  // теперь ptr_shape указывает на объект square
ptr_shape->display( ); // динамическое связывание с display() из класса Квадрат
rectangle.display( );  // статическое связывание с display() из класса Прямоугольник
```

Функция `display()` в определении родительского класса `Форма` установлена в ноль. Так указывается, что этот элемент-функция является чисто виртуальной функцией, не имеющей тела (которое может быть вызвано). Тело должно быть переопределено в классах-потомках.

Цель чисто виртуальной функции — задать идею, интерфейс функции, без раскрытия реализации. Класс, включающий чисто виртуальную функцию, является абстрактным классом. Объект абстрактного класса создать нельзя. Иначе говоря, абстрактный тип данных не может иметь конкретные объекты, а используется для представления идеи типа. Подклассы же такого типа способны иметь объекты.

Динамическое связывание позволяет клиентам использовать элементы вида `display()` даже до написания всех версий `display()`. Новые классы-потомки могут добавляться спустя годы. При этом не требуются изменения в программах, которые применяют динамически связываемые элементы. Это очень полезное свойство объектно-ориентированного подхода.

ООП на языке Ада

Расширяемые типы

Основная цель расширяемых типов — обеспечить повторное использование существующих программных элементов (без необходимости перекомпиляции и перепроверки). Они позволяют объявить новый тип, который уточняет существующий родительский тип наследованием, изменением или добавлением как существующих компонентов, так и операций родительского типа. В качестве расширяемых типов используются *теговые типы* (разновидность комбинированного типа).

Рассмотрим построение иерархии геометрических объектов. На вершине иерархии находится точка, имеющая два атрибута (координаты *X* и *Y*):

```
type Точка is tagged
    record
        x_Коорд : Float;
        y_Коорд : Float;
    end record;
```

Другие типы объектов можно произвести (прямо или косвенно) от этого типа.

Например, можно ввести новый тип, наследник точки:

```
type Окружность is new Точка with  -- новый теговый тип;
    record
        радиус : Float;
    end record;
```

Данный тип имеет три атрибута: два атрибута (координаты `x` и `y`) унаследованы от типа `Точка`, а третий атрибут (`радиус`) нами добавлен. Дочерний тип `Окружность` наследует все операции родительского типа `Точка`, причем некоторые операции могут быть переопределены. Кроме того, для дочернего типа могут быть введены новые операции.

Классы

В языке Ада нет синтаксической конструкции «класс». Для моделирования конструкции «класс» здесь применяют конструкцию «пакет». Пакет — это элегантный способ инкапсуляции программного кода и данных, взаимодействующих друг с другом, в единый модуль. Пакет может экспортировать один или несколько пользовательских типов вместе с их примитивными операциями. Примитивная операция определяется для типа (в части аргументов и результата) и объявляется вместе с типом в одной спецификации пакета.

Правила моделирования класса

1. Имени класса придается префикс `Класс_`.
2. Пакет имеет единственный теговый приватный тип, который получает имя класса и используется для объявления экземпляров класса. Вследствие этого все экземпляры класса будут разделять одинаковую структуру и поведение.
3. Процедуры и функции используются для определения поведения класса. Первым параметром для процедуры или функции является экземпляр класса.

4. Реализация приватного типа определяется как комбинированный расширяемый тип. (Это дает возможность расширять количество полей данных в классах-наследниках, а также количество операций.) Компоненты комбинированного типа определяют структуру данных класса.

   Например, класс `Счет` задается в виде следующего пакета:

```
package Класс_Счет is
    type Счет is tagged private; -- осн. тип класса
               -- используется для объявления экземпляров
    subtype Деньги is Float; -- вспомог. типы
    subtype РДеньги is Float range 0.0.. Float'LAST;
    -- используются в сообщениях,  посылаемых в экз.
    -- объявления методов класса:
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет;
                    Сумма : in РДеньги );
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги );
    function баланс ( the : Счет ) return Деньги;
    private    -- скрытое представление класса
        type Счет is tagged record
            остаток : Деньги := 0.00; -- сумма на счету
        end record;
end  Класс_Счет;
-- Тело пакета-класса включает реализацию его методов:
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;
package body Класс_Счет is
    procedure заявить ( the : in Счет ) is
    begin
        Put ("Тек. состояние : Сумма на вкладе $");
        Put ( the.остаток, Aft => 2, Exp => 0);
        New_Line ( 2 );
    end заявить;
    procedure положить ( the : in out Счет;
                    сумма : in РДеньги ) is
    begin
        the.остаток := the.остаток + сумма;
    end положить;
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги ) is
    begin
        if the.остаток >= сумма then
            the.остаток := the.остаток - сумма;
            принимать := сумма;
        else принимать := 0.00;
        end if;
    end снять;
    function баланс ( the : Счет ) return Деньги is
    begin
        return  the.остаток;
    end баланс;
end Класс_Счет;
```

Видим, что для доступа к полю данных `остаток`, содержащемуся в экземпляре `Счета`, используется составное имя `the.остаток`. Например, в функции `баланс` результат возвращается с помощью оператора

```
return  the.остаток;
```

Составное имя используется для доступа к атрибуту экземпляра комбинированного типа. В этом случае экземпляром типа запись является объект `the`, а атрибутом объекта — поле `остаток`.

К объектам класса (типа) `Счет` применимы следующие операции:

- обработка (предвыполнение) при создании объекта;
- присвоение значения экземпляра другому экземпляру такого же типа;
- сравнение экземпляров класса на эквивалентность и неэквивалентность;
- заданные методы (чтение и изменение внутреннего состояния возможно только в результате действия методов класса).

Если мы объявили объект класса `Счет`

```
мой_Счет: Класс_Счет.Счет;
```

то можно использовать две формы обращения к его операциям:

```
1.    положить (мой_Счет, деньги);
2.    мой_Счет.положить (деньги);
```

В первой форме в качестве первого аргумента вызываемой операции указывается имя объекта (которому принадлежит эта операция). Во второй форме сообщение начинается с указания имени адресуемого объекта, поэтому из списка аргументов операции имя объекта изымается. Вторая форма соответствует традиционному стилю записи сообщений, принятому в объектно-ориентированном программировании.

Абстрактные классы и интерфейсы

Абстрактный класс — это описание (спецификация) будущих возможностей, которые будут обеспечены в дальнейшем производными классами. Абстрактный класс не имеет тела, то есть реализации.

Основным элементом абстрактного класса является абстрактный расширяемый тип. Этот тип задает только имя. Он не имеет полей, то есть экземплярных атрибутов, и поэтому записывается в публичной части спецификации в виде:

```
type Абстрактный_Счет is abstract tagged null record;
```

Примитивными операциями абстрактного типа (методами абстрактного класса) являются абстрактные процедуры и функции. Абстрактные процедуры и функции также не имеют тел, их назначение — объявить спецификации будущих конкретных методов.

В качестве примера приведем абстрактный класс банковского счета:

```
package Класс_Абстрактный_Счет is
    type Абстрактный_Счет is abstract tagged null record;
    subtype Деньги is Float;  -- вспомогат. подтипы
    subtype Рденьги is Float range 0.0 .. Float'Last;
    procedure заявить ( the : in Абстрактный_Счет ) is abstract;
    procedure положить ( the : in out Абстрактный_Счет; сумма : in Рденьги ) is
abstract;
    procedure снять ( the : in out Абстрактный_Счет; сумма : in РДеньги;
        принимать : out РДеньги ) is abstract;
    function баланс ( the : in Абстрактный_Счет ) return Деньги is abstract;
end Класс_Абстрактный_Счет;
```

Сам по себе абстрактный класс и абстрактный тип нельзя использовать для объявления объектов. Однако абстрактный класс можно применить для производства (путем наследования) конкретных типов банковского счета, например:

```
with Класс_Абстрактный_Счет;
use Класс_Абстрактный_Счет;
package Класс_Счет is
    type Счет is new Абстрактный_Счет with private;
    subtype Деньги is Класс_Абстрактный_Счет.Деньги;
    subtype Рденьги is Класс_Абстрактный_Счет.РДеньги;
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет; сумма : in Рденьги );
    procedure снять ( the : in out Счет; сумма : in РДеньги; принимать : out
                                                                РДеньги );
    function баланс ( the : in Счет ) return Деньги;
private
    type Счет is new Абстрактный_Счет with
        record
            остаток : Деньги := 0.00;  -- сумма на счету
        end record;
end Класс_Счет;
```

ПРИМЕЧАНИЕ

Подтипы Деньги и РДеньги объявлены для того, чтобы сделать их видимыми для клиентов класса. Если этого не сделать, то клиенты должны будут ссылаться на Класс_Абстрактный_Счет (с помощью with и use). Тело (реализация) этого класса аналогично телу класса Счет из предыдущего подраздела.

В свою очередь, конкретный класс `Счет` может использоваться как родительский класс в производстве нового класса. Например, может быть произведен счет, по которому операцию `снять` разрешено выполнять только три раза в неделю:

```
with Класс_Счет; use Класс_Счет;
package Класс_Огр_Счет is
    type Огр_Счет is new Счет with private;
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
                принимать : out РДеньги );
    procedure сброс ( the : in out Огр_Счет);
private
    снять_За_Неделю : Natural := 3;
    type Огр_Счет is new Счет with
        record
            снятия : Natural := снять_За_Неделю;
        end record;
end Класс_Огр_Счет;
```

ПРИМЕЧАНИЕ

В данном классе переопределяется родительский метод снять. Новый метод `сброс` применяется для сброса количества снятий, которые могли быть выполнены на текущей неделе.

Реализация этого класса записывается в виде:

```
package body Класс_Огр_Счет is
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
                      принимать : out РДеньги ) is
    begin
        if the.снятия > 0 then  -- проверка ограничения
            the.снятия := the.снятия - 1;
            снять (Счет(the), сумма, принимать);
                -- вызов родительского метода
        else
            принимать := 0.00;    -- извините
        end if;
    end снять;
    procedure сброс ( the : in out Огр_Счет) is
    begin
        the.снятия := снять_За_Неделю;
    end сброс;
end Класс_Огр_Счет;
```

Использование класса `Огр_Счет` проиллюстрируем следующей программой:

```
with Класс_Счет, Класс_Огр_Счет;
use Класс_Счет, Класс_Огр_Счет;
procedure Main is
    петр : Огр_Счет;
    получить : Деньги;
begin
    петр.положить (700.00);
    петр.заявить;   -- количество денег на счету
    петр.снять (150.00, получить); -- снятие денег
    петр.снять (70.00, получить); -- снятие денег
    петр.снять (20.00, получить); -- снятие денег
    петр.снять (15.00, получить);
            -- отказ - превышен лимит
    петр.заявить; -- количество денег на счету
end Main;
```

ПРИМЕЧАНИЕ

Спецификаторы with и use для Класс_Счет обеспечивают прямую видимость подтипа Деньги. Вспомним, что подтип Деньги объявлен в классе Счет и невидим в классе Огр_Счет. Можно отказаться от спецификатора use Класс_Счет, если в программе Main для переменной `получить` применить объявление получить:Класс_Счет.Деньги.

Если абстрактный класс вообще не имеет ни конкретных операций, ни компонентов данных, он может быть объявлен как интерфейс. Интерфейс — это предельно ограниченный вариант абстрактного типа. В нем нет объявлений полей данных, а операции объявлены или абстрактными, или нулевыми. Вместе с тем интерфейс — важнейший механизм обеспечения множественного наследования.

Объявление интерфейса записывается в публичной части спецификации в виде:

```
type My_Abstract is Interface;
```

Приведем ряд примеров. Множественное наследование можно обеспечить, объявив родителями конкретный тип `My_Parent` и два интерфейса `Int1` и `Int2`:

```
type My_Child is new My_Parent and Int1 and Int2 with …
```

Можно составить новый интерфейс из двух имеющихся:

```
type Int3 is Interface and Int1 and Int2;
```

Надклассовые типы

Для обеспечения полиморфизма в языке Ада введено понятие надклассового типа.

Для каждого тегового типа `T` автоматически объявляется надклассовый тип `T'Class`. Значения `T'Class` являются объединением значений самого типа `T` и всех производных от него типов. Сам тип `T` считается корневым типом для дерева наследования, которое представляет `T'Class`. Значение любого дочернего от `T` типа может быть неявно преобразовано к надклассовому типу `T'Class`.

Например, если наследниками тегового типа `Комната` являются типы `Кладовая` и `Офис`, а тип `Офис`, в свою очередь, имеет наследника `Канцелярия`, то все эти типы образуют надклассовый тип `Комната'Class`.

Значение любого типа помещения может быть неявно преобразовано в значение типа `Комната'Class`. В свою очередь, надклассовый тип `Офис'Class` включает в себя конкретные типы `Офис` и `Канцелярия`, а надклассовый тип `Кладовая'Class` — только конкретный тип `Кладовая`.

Каждый объект надклассового типа имеет свой тег, который задает его конкретный тип во множестве других типов дерева наследования. Единственность тега объясняется его происхождением: тег — это элемент конкретного тегового типа. Тег объекта может быть явно выделен с помощью атрибута `'Tag`. Например, если `b304` и `b306` — это объекты типа `Комната'Class`, то можно записать:

```
if b304'Tag = b306'Tag then
    Put ("Аудитории имеют один и тот же тип помещения");
    New_Line;
end if;
```

Надклассовый тип `T'Class` считается неограниченным типом, так как заранее нельзя предугадать размер объекта этого типа. Поэтому поступают следующим образом:

- объявляют объект надклассового типа;
- инициализируют объект, вследствие чего он ограничивается с помощью тега.

Говорят, что надклассовые типы являются полиморфными. Когда сообщение (например, `Описать`) посылается в объект надклассового типа `Комната'Class`, компилятор в период компиляции не знает, какой метод будет выполняться. Решение об исполняемом методе принимается в период выполнения (путем анализа тега объекта). В терминологии языка Ада динамическое связывание между объектом и посылаемым сообщением называется *диспетчированием времени выполнения*.

Наследование от родового класса

В главе 15 обсуждался родовой АТД для стека и его родовое дочернее расширение. Для превращения их в классы достаточно заменить использованный там обычный комбинированный тип на расширяемый теговый тип. В результате такой замены получим:

```
generic
    type T is private;  -- любой нелимитир. тип
    Size : Positive := 4;
                -- указан тип и значение по умолчанию
package Class_Stack is
    type Stack is tagged private;
    Stack_Error : exception;
    procedure Reset ( The : in out Stack );
    procedure Push ( The : in out Stack; Item : in T );
    procedure Pop ( The : in out Stack; Item : out T );
private
    type Stack_Index is new Integer range 0 .. Size;
    subtype Stack_Range is Stack_Index
                range 1 .. Stack_Index ( Size );
    type Stack_Array is array ( Stack_Range ) of T;
    type Stack is tagged record
        Elements : Stack_Array;   -- массив элементов
        Tos : Stack_Index := 0;   -- указатель на вершину
    end record;
end Class_Stack;
```

Тело родового класса содержит реализацию его методов:

```
package body Class_Stack is
    procedure Reset ( The : in out Stack ) is
    begin
        The.Tos := 0;
                -- установить указатель  в нуль (нет элементов)
    end Reset;
    procedure Push ( The : in out Stack; Item : in T ) is
    begin
        if The.Tos /= Stack_Index ( Size ) then
                        -- проверка заполнения
            The.Tos := The.Tos + 1;  -- указатель вверх
            The.Elements (The.Tos) := Item; -- запись элемента
        else
            raise Stack_Error;    -- ошибка переполнения
        end if;
    end Push;
procedure Pop ( The : in out Stack; Item : out T ) is
    begin
        if The.Tos > 0 then
            Item := The.Elements (The.Tos ); --счит.верхн.элемен.
            The.Tos := The.Tos - 1;  -- указатель вниз
else
            raise Stack_Error;
                -- ошибка удаления из пустого стека
        end if;
    end Pop;
end Class_Stack;
generic
package Class_Stack.Additions is
    function Top ( The : in Stack ) return T;
    function Items ( The : in Stack ) return Natural;
private
end Class_Stack.Additions;
package body Class_Stack.Additions is
    function Top ( The : in Stack ) return T is
```

```
    begin
        return The.Elements ( The.Tos );
    end Top;
    function Items ( The : in Stack ) return Natural is
    begin
        return Natural ( The.Tos );
    end Items;
end Class_Stack.Additions;
```

Положим, что теперь к родовому классу `Class_Stack` и его родовой дочери `Class_Stack.Additions` нужно добавить метод.

Метод	Обязанность
Max_Depth	Вернуть максимальную глубину, которая достигнута стеком

Для решения этой задачи используем механизм наследования. Спецификацию наследника представим в виде:

```
with Class_Stack, Class_Stack.Additions;
generic
    type T is private;    -- любой нелимитирован. тип
    Size : Positive := 4; -- указан тип и значение по умолчанию
package Class_Best_Stack is
            package Class_Stack_D is new Class_Stack ( T, Size );
            package Class_Stack_D_Additions is
                    new Class_Stack_D.Additions;
        -- этот экземпляр становится независимым от родителя
            type Best_Stack is new Class_Stack_D.Stack with private;
        -- переопределяем метод
            procedure Push ( The : in out Best_Stack; Item : in T );
        -- вводим новый метод
            function Max_Depth ( The : in Best_Stack )
                        return Natural;
private
    type Best_Stack is new Class_Stack_D.Stack with
                record
                    Depth : Natural := 0;  -- новый атрибут
                end record;
end Class_Best_Stack;
```

ПРИМЕЧАНИЕ

1. Таким образом, конкретизацию базового класса Stack и его родовой дочери мы разместили в теле класса-наследника.
2. Метод Push переопределен для того, чтобы он мог фиксировать максимальную достигнутую глубину.

Реализация класса-наследника выглядит так:

```
package body Class_Best_Stack is
    procedure Push ( The : in out Best_Stack; Item : in T ) is
                Dep : Natural;
        begin
            Dep := Class_Stack_D_Additions.Items
                        (Class_Stack_D.Stack(The));
```

продолжение ⇩

```
            if Dep > The.Depth then
                    The.Depth := Dep;
            end if;
            Class_Stack_D.Push (Class_Stack_D.Stack (The),Item);
        end Push;
    function Max_Depth ( The : in Best_Stack ) return Natural is
        begin
            return The.Depth;
        end Max_Depth;
end Class_Best_Stack;
```

Конкретизация класса `Class_Best_Stack` для чисел типа `Positive` выполняется по объявлению:

```
with Class_Best_Stack;
    package Class_Best_Stack_Pos is
            new Class_Best_Stack ( Positive, 15 );
```

Тестирование конкретизированного класса можно осуществить с помощью программы:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Class_Best_Stack_Pos;
use Ada.Text_IO, Ada.Integer_Text_IO, Class_Best_Stack_Pos;
procedure Main is
    Numbers : Best_Stack;
    Res : Positive;
begin
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Push ( Numbers, 40 );
    Push ( Numbers, 20 );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Push ( Numbers, 50 );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
    Pop ( Numbers, Res );
    Put ("Максимальная глубина: ");
    Put_Line ( Max_Depth ( Numbers ));
end Main;
```

Данная программа выводит на экран следующие результаты:

```
Максимальная глубина: 0
Максимальная глубина: 2
Максимальная глубина: 3
Максимальная глубина: 3
```

ООП на языке Java

Язык Java поддерживает работу как с объектами, так и с величинами простых скалярных типов (предусмотрен логический, символьный и числовые типы). Перечисления и массивы в Java являются объектами. Каждому простому типу сопоставлен класс-оболочка. Начиная с пятой версии, здесь предусмотрен механизм автоматической упаковки/распаковки. Упаковка (auto-boxing) — это процесс автоматического сохранения величин простого типа (такого как `int`, или `double`) в эквивалентном объекте-оболочке. Распаковка (auto-unboxing) является обратным

действием, она приводит к автоматическому извлечению значения из упакованного объекта. Иными словами, речь идет о неявном преобразовании типов значений.

Благодаря упаковке нет нужды вручную формировать объект для хранения значения простого типа. Достаточно присвоить это значение ссылке на объект класса-оболочки. Язык Java автоматически сгенерирует этот объект. Например, для создания объекта типа `Integer`, хранящего значение 77, записывается:

```
Integer iObject = 77; // автоматически упаковывает значение типа int
```

Для распаковки объект просто присваивается переменной простого типа. Например, для распаковки объекта `iObject` выполняется следующее:

```
int i = iObject; // автораспаковка
```

Все классы в языке Java должны быть подклассами корневого класса `Object` или потомка этого класса. Все объекты размещаются в куче, с помощью оператора `new`. Удаления объектов из кучи производится посредством сборки мусора. Сборщик мусора лишь восстанавливает динамическую память из-под объектов, но не способен решать сопутствующие проблемы. Например, если объект в куче задействовал какой-то внешний ресурс (файл или внешнюю разделяемую память), сборка мусора не сможет его освободить. В подобных ситуациях предлагается использование специального метода `finalize()`, который имеет некоторое сходство с деструктором языка C++.

Метод `finalize()` неявно вызывается сборщиком мусора, когда он собирается освободить память из-под конкретного объекта. Проблема заключается в том, что время финализации не может быть предсказано или задано принудительно. Альтернативой финализации может стать метод, формирующий требование о возвращении ресурсов. Правда, все клиенты объекта должны знать о подобном методе и вовремя вызывать его.

Единичное наследование

В языке Java метод может быть определен со спецификатором `final`, что вводит запрет на его переопределение в любом классе-наследнике. Если зарезервированное слово `final` появляется в определении класса, то у такого класса не может быть классов-наследников. Язык Java использует аннотацию `@Override`, которая информирует компилятор о необходимости проверки переопределения последующим методом некоторого метода в классе-предке. При отрицательном результате компилятор формирует сообщение об ошибке. Как и язык C++, Java требует, чтобы перед вызовом конструктора подкласса был вызван конструктор родительского класса. Если же в родительский конструктор нужно передавать параметры, то вызов этого конструктора должен быть записан явно, как показано в следующем примере:

```
super(77, true);
```

Если же явный вызов родительского конструктора отсутствует, компилятор сам вставит вызов родительского конструктора без параметров. Java не поддерживает приватное и защищенное наследование, реализованное в языке C++. Дело в том, что разработчики языка считают: подклассы должны быть подтипами, чему противоречит приватное и защищенное наследование. Ранние версии языка Java включали коллекцию `Vector`, содержащую множество методов для обработки данных. Кроме

того, предлагался подкласс вектора — `Stack`, добавлявший методы для операций `push()` и `pop()`. Поскольку в языке нет приватного наследования, все методы вектора были видимы в классе `Stack`, поэтому объекты допускали использование целого ряда операций, некорректных для стека.

В языке C++ для обеспечения динамического связывания метод должен быть определен как виртуальный. В языке Java сообщения динамически связываются со всеми методами. Исключением является метод с пометкой `final`, который не может переопределяться и связывание с которым является статическим. Статическое связывание применяется также, если метод помечен как `static` или `private`; любая из этих пометок запрещает переопределение.

Смешанное наследование

Язык Java непосредственно поддерживает только единичное наследование. Однако имеется разновидность абстрактного класса, называемая интерфейсом, которая обеспечивает частичную поддержку множественного наследования. Определение интерфейса подобно определению класса, однако содержит только именованные константы и спецификации методов (без реализации). Оно не может включать конструкторы или не абстрактные методы. Следовательно, интерфейс лишь декларирует имена, определяя спецификацию класса. Класс не может наследовать от интерфейса, ему позволено лишь реализовать интерфейс. Более того, класс может реализовать любое количество интерфейсов. При реализации интерфейса класс обязан реализовать все методы, чьи спецификации (не тела) прописаны в определении интерфейса.

Интерфейс можно использовать для моделирования множественного наследования. Подкласс может быть наследником одного суперкласса и реализовать несколько родительских интерфейсов. Иногда это называют *смешанным наследованием*, поскольку константы спецификации методов из интерфейсов смешиваются с методами и данными, унаследованными от суперкласса.

Очень интересной особенностью интерфейсов является то, что они обеспечивают некоторую разновидность полиморфизма. Суть в том, что интерфейсы могут трактоваться как типы. К примеру, некоторый формальный параметр метода определяется как интерфейс. Такому формальному параметру соответствует фактический параметр любого класса, который реализует интерфейс. Следовательно, метод становится полиморфным.

Переменная, не рассматриваемая как параметр, тоже может объявляться с типом, в роли которого выступает интерфейс. Такая переменная может ссылаться на любой объект любого класса, который реализует интерфейс.

Одна из проблем множественного наследования состоит в получении от двух родительских классов двух публичных методов с одинаковым именем и протоколом. При использовании интерфейсов эта проблема исчезает. Хотя подкласс, реализующий интерфейс, и должен обеспечить полные определения всех методов интерфейса, однако при совпадении имени и протокола у методов суперкласса и интерфейса вовсе не требуется повторно реализовывать этот метод. Таким образом, конфликт из-за имени метода, возможный при множественном наследовании, исчезает при одновременном использовании единичного наследования и реализации интерфей-

сов. Более того, полностью исчезают конфликты по имени переменной, поскольку интерфейсы не могут определять переменные.

Интерфейсы не замещают множественного наследования, так как во множественном наследовании речь идет о повторном использовании кода реализации, а интерфейсы такой возможности совсем не обеспечивают. Это существенное различие, поскольку повторное использование кода является одним из главных преимуществ наследования. Язык Java лишь предлагает определенный способ борьбы с недостатками наследования. Один из реализуемых интерфейсов можно заменить абстрактным классом, содержащим наследуемый код, тем самым обеспечивая некоторое повторное использование кода.

Одна из проблем применения интерфейсов для представления множественного наследования состоит в следующем: если класс пытается реализовать два интерфейса, в каждом из которых присутствуют спецификации методов с одинаковым именем и протоколом, то создать две реализации просто нельзя.

В качестве примера интерфейса обсудим метод `sort()` стандартного библиотечного класса `Array`. Любой класс, использующий этот метод, должен обеспечить реализацию метода для сравнения сортируемых элементов. Родовой интерфейс `Comparable` обеспечивает протокол для данного метода сравнения по имени `compareTo`. Программный код для интерфейса `Comparable` имеет следующий вид:

```
public interface Comparable <T> {
    public int compareTo(T b);
}
```

Метод `CompareTo` в качестве параметра принимает ссылку на какой-то объект и сравнивает его с текущим объектом, возвращая отрицательное, положительное или нулевое целое число, если, соответственно, текущий объект меньше или больше параметра или равен ему. Класс, который реализует интерфейс `Comparable`, может сортировать содержимое любого массива из объектов того типа, который подставляется реализующим методом вместо родового типа `T`.

Кроме интерфейсов, язык Java предлагает абстрактные классы, где абстрактные методы описываются заголовками с использованием зарезервированного слова `abstract`. Абстрактный класс также помечается спецификатором `abstract`. Как обычно, у абстрактных классов не может быть экземпляров.

Вложенные классы

Язык Java предлагает несколько разновидностей вложенных классов, которые скрыты от всех классов, размещенных внутри некоторого пакета (за исключением объемлющего класса, где находится вложенный класс). Не статические классы, непосредственно вкладываемые в другой класс, называются *внутренними классами*.

Каждый экземпляр внутреннего класса должен иметь неявный указатель на экземпляр объемлющего класса. Это позволяет всем методам вложенного класса иметь доступ ко всем элементам объемлющего класса, включая приватные элементы. Статические вложенные классы не обладают таким указателем и лишены доступа к элементам объемлющего класса. Следовательно, статические вложенные классы в языке Java подобны вложенным классам в C++.

Хотя это и странно для языка со статической областью видимости, элементы внутреннего класса, даже приватные элементы, доступны во внешнем классе. Эту возможность обеспечивает переменная, которая ссылается на объект внутреннего класса. Например, допустим, что внешний класс создает экземпляр внутреннего класса:

```
theInner = this.new Inner();
```

Если внутренний класс определил переменную `total`, из внешнего класса к ней можно обратиться по ссылке `theInner.total`.

Экземпляр вложенного класса может существовать только внутри экземпляра своего объемлющего класса. Вложенные классы могут быть анонимными. Анонимные классы имеют сложный синтаксис (он обсуждался в разделе «Природа наследования»), который является сокращенной формой определения класса, используемого только в одном месте.

Локальный вложенный класс определяется в методе объемлющего класса. Локальные вложенные классы никогда не определяются со спецификатором доступа (`private` или `public`). Их область видимости всегда ограничена пределами объемлющего класса. Метод в локальном вложенном классе может обращаться к переменным объемлющего класса и к переменным с меткой `final` из метода, где определен сам локальный вложенный класс. Элементы локального вложенного класса видимы только в методе, где определен этот класс.

ООП на языке C#

Для поддержки объектно-ориентированного подхода язык C# предоставляет классы и структуры. Классы очень похожи на классы языка Java, а структуры считаются их облегченными прообразами, которые размещаются в стеке и не поддерживают наследования. Более подробно структуры обсуждались в главе 15.

Наследование

Синтаксис определения подкласса в языке C# заимствован из C++. Например:

```
public class Child : Parent { . . . }
```

Метод, унаследованный от класса-родителя, может *заменяться* в классе-потомке новой версией, его новое определение имеет пометку `new`. При обычном доступе новый метод скрывает унаследованный родительский метод с тем же именем. Тем не менее родительская версия может быть вызвана при указании префикса `base`:

```
base.Display();
```

Метод потомка, переопределяющий родительский метод, должен иметь пометку `override`. Язык C# поддерживает интерфейсы в том же стиле, который принят в языке Java.

Динамическое связывание

Для поддержки динамического связывания сообщения с методами в C# и родительский метод и соответствующие ему методы в производных классах должны быть

специально помечены. В родительском классе метод помечается словом `virtual` (как в C++). Методы производных классов, переопределяющие виртуальный родительский метод и имеющие такое же имя и протокол, помечаются словом `override`. К примеру, версия C# для иерархии классов, где корневым является класс `Форма` (рассматривалась в разделе «ООП на языке C++»), принимает следующий вид:

```
public class Форма {
    public virtual void Display() { . . . }
    . . .
}
public class Круг : Форма {
    public override void Display() { . . . }
    . . .
}
public class Прямоугольник : Форма {
    public override void Display() { . . . }
    . . .
}
public class Квадрат : Прямоугольник {
    public override void Display() { . . . }
    . . .
}
```

Язык C# содержит абстрактные методы, подобные абстрактным методам языка C++, но использующие другой синтаксис. Приведем пример абстрактного метода в языке C#:

```
abstract public void Display();
```

Класс, включающий хотя бы один абстрактный метод, считается абстрактным классом. Каждый абстрактный класс должен быть помечен словом `abstract`. Как обычно, абстрактный класс не имеет экземпляров. Отсюда следует, что любой подкласс абстрактного класса, который намерен иметь экземпляры, должен реализовать все абстрактные унаследованные методы.

Как и в языке Java, все классы в C# произведены, в конечном счете, от единого корневого класса `Object`. Класс `Object` определяет набор методов, включая `ToString`, `Finalize` и `Equals`, которые наследуются всеми типами в C#.

Вложенные классы

Класс в C#, который непосредственно вложен в класс, ведет себя подобно статическому вложенному классу языка Java (который подобен вложенному классу в C++). Как и C++, язык C# не поддерживает вложенные классы, которые ведут себя как не статические вложенные классы языка Java.

ООП на языке Ruby

Ruby — чисто объектно-ориентированный язык с динамической типизацией. Практически все сущности в языке являются объектами, а все вычисления инициируются пересылками сообщений. Конечно, в программах есть выражения из инфиксных операций, внешне похожие на обычные выражения, но рассматривать эти выражения

следует через призму пересылки сообщений. Как и в языке Smalltalk, запись `x + y` выполняется посылкой сообщения `+` объекту `x`, с передачей объекта `y` в качестве параметра. Иначе говоря, `x + y` реализуется как `x.+ y`.

Определения классов в Ruby отличаются от определений в языках C++ и Java прежде всего тем, что являются исполняемыми сущностями, а не просто статическими описаниями. В силу этого они остаются открытыми в ходе вычислений. Время от времени программа может добавлять в класс элементы, обеспечивая вторичные определения класса с новыми элементами. В процессе вычислений текущее определение класса является объединением всех выполненных определений классов. Определения методов также имеют исполняемую форму, что позволяет программе прямо в ходе вычислений задавать требуемые версии определения методов (путем простой подстановки определений в ветви `then` и `else` конструкции ветвления).

Все переменные в Ruby считаются ссылками на объекты, не имеющие типа. Имена экземплярных переменных начинаются со знака `@`.

В отличие от других языков, в Ruby управление доступом к данным отличается от доступа к методам. По умолчанию для всех экземпляров данных задан приватный доступ, который нельзя отменить. Для внешнего доступа к экземплярной переменной должен быть предоставлен (определен) метод доступа. В качестве примера рассмотрим следующее определение класса:

```
class TheClass
# Конструктор
    def initialize
    @four = 4
    @five = 5
    end
# Метод чтения для @four
    def four
    @four
    end
# Метод записи для @four
    def four =(the_four)
    @four = the_four
    end
end # конец класса TheClass
```

Знак равно `=`, добавляемый к имени метода записи, означает, что к переменной применимо присваивание. Следовательно, в именах всех методов записи присутствует знак «равно». Тело метода чтения иллюстрирует принятое в Ruby обозначение для возврата значения вычисленного выражения (в отсутствии оператора `return`). В этом случае возвращается значение `@four`. Поскольку методы чтения и записи достаточно популярны, Ruby предлагает для них сокращенные формы записи. Если требуется, чтобы класс обеспечил методы чтения для двух экземплярных переменных `@four` и `@five`, эти методы можно определить единичным оператором класса:

```
attr_reader :four, :five
```

В действительности `attr_reader` является вызовом функции, использующим фактические параметры `:one` и `:two`. Префикс «двоеточие» перед именем переменной говорит о том, что требуется именно переменная, а не разыменование объекта, на который она ссылается.

Функция для создания методов записи называется `attr_writer`. Эта функция имеет такой же профиль параметров, как и `attr_reader`.

Названия функций для создания методов чтения и записи обусловлены тем, что они обеспечивают протокол доступа к атрибутам объектов какого-то класса. Следовательно, атрибуты класса определяют интерфейс данных (благодаря методам доступа данные становятся публичными) для объектов класса.

Объекты в Ruby создаются с помощью оператора `new`, который неявно вызывает конструктор. Обычный конструктор в классах Ruby называется `initialize`. Конструктор подкласса может инициализировать элементы данных родительского класса, для которых определены методы записи. Необходимый вызов начинается со слова `super`, за которым записываются фактические параметры (начальные значения). В итоге вызывается одноименный метод родительского класса (имя родительского класса повторяет имя метода, где появился вызов `super`).

Если имя переменной в классе начинается с двух знаков `@@`, то переменная считается приватной для класса и его экземпляров. Эту приватность изменить нельзя. В отличие от глобальных и экземплярных переменных, переменные класса должны быть инициализированы перед использованием.

Наследование

Подклассы определяются в Ruby с помощью символа меньше `<`, а не символа двоеточия, применяемого в C++. Например:

```
class Child < Parent
```

Отметим особенность метода управления доступом в Ruby: в подклассе доступ может быть изменен простым вызовом функций управления доступом. Это означает: два подкласса можно определить так, что объекты одного подкласса получат доступ к родительскому методу, а объектам другого подкласса доступ будет запрещен. Кроме того, в подклассе публично доступный родительский метод можно превратить в приватно доступный метод. Такой подкласс перестает быть подтипом.

Модули в Ruby обеспечивают инкапсуляцию пространства имен, такая возможность достаточно часто используется для определения библиотек функций. Самым интересным аспектом модулей считается то, что их функции могут быть непосредственно доступны из классов. В классе доступ к модулю задается оператором `include`:

```
include Math
```

В результате включения модуля класс получает указатель на модуль и наследует все функции, определенные в модуле. При включении в класс модуль становится представителем суперкласса в классе. Говорят, что такой модуль обеспечивает микширование.

Динамическое связывание

Динамическое связывание в Ruby подобно механизму связывания в языке Smalltalk. Переменные не типизированы, а считаются ссылками на объекты любого класса. Следовательно, все переменные полиморфны, а все связывания методов являются динамическими.

Реализация объектно-ориентированных классов и объектов

Рассмотрим два самых важных вопроса реализации объектно-ориентированных конструкций: организацию памяти для сохранения объектов и динамическое связывание сообщений с методами объектов.

Организация памяти для сохранения объекта

Назовем структуру, необходимую для сохранения объекта в памяти, структурой экземпляра класса СЭК. Структура СЭК статична, поэтому она формируется в период компиляции и используется как шаблон при создании экземпляров класса. Каждый класс имеет свою собственную СЭК. Подклассы могут расширять структуру СЭК класса-родителя, добавляя поля для новых элементов потомка.

Доступ ко всем атрибутам (полям данных) объекта подобен доступу к полям записей: используются конкретные смещения от начала структуры для экземпляра класса. В результате такой доступ столь же эффективен, как и доступ к полям записей.

Динамическое связывание сообщений с методами

Методы класса, связывание с которыми осуществляется статически, не нужно включать в структуру СЭК данного класса. Однако для динамически связываемых методов приходится вводить дополнительные поля. В этих полях можно хранить простые указатели на коды реализации методов. Значения указателей устанавливаются во время создания объекта. Далее вызовы метода связываются с соответствующим кодом через его указатель в СЭК. Недостаток такого подхода состоит в том, что *каждый* объект должен сохранять указатели на все динамически связываемые методы, которые могут вызваться из этого объекта.

Заметим, что список динамически связываемых методов, вызываемых из объектов, одинаков для всех экземпляров конкретного класса. Таким образом, список таких методов можно хранить только в одном месте. Отсюда вывод: в структуре СЭК объекта следует иметь только один указатель на список, обеспечивающий нахождение вызываемых методов. Структуру памяти для хранения этого списка обычно называют таблицей виртуальных методов ТВМ. Адрес нужного метода находится по конкретному смещению от начала таблицы ТВМ. Полиморфные переменные родительского класса всегда ссылаются на структуру СЭК объекта нужного типа, поэтому получение нужной версии динамически связываемого метода гарантируется. Рассмотрим фрагмент на языке Java, в котором все методы предполагаются динамически связываемыми:

```
public class Parent {
    public int x, y;
    public void display() { . . . }
    public int square() { . . . }
}
```

```
public class Child extends Parent {
    public int m, n;
    public void display() { . . . }
    public void move() { . . . }
}
```

Структуры СЭК для классов `Parent` и `Child`, а также таблицы их виртуальных методов представлены на рис. 16.24. Заметим, что указатель на метод `square()` из таблицы для `Child` указывает на реализацию в классе `Parent`. Причина в том, что класс `Child` не переопределяет этот родительский метод, поэтому, когда клиент `Child` будет вызывать `square()`, реализация данного метода берется из `Parent`. С другой стороны, указатели на `display()` и `move()` из виртуальной таблицы для `Child` указывают на их реализации из `Child`. Метод `display()` переопределен в `Child`, а `move()` просто добавлен.

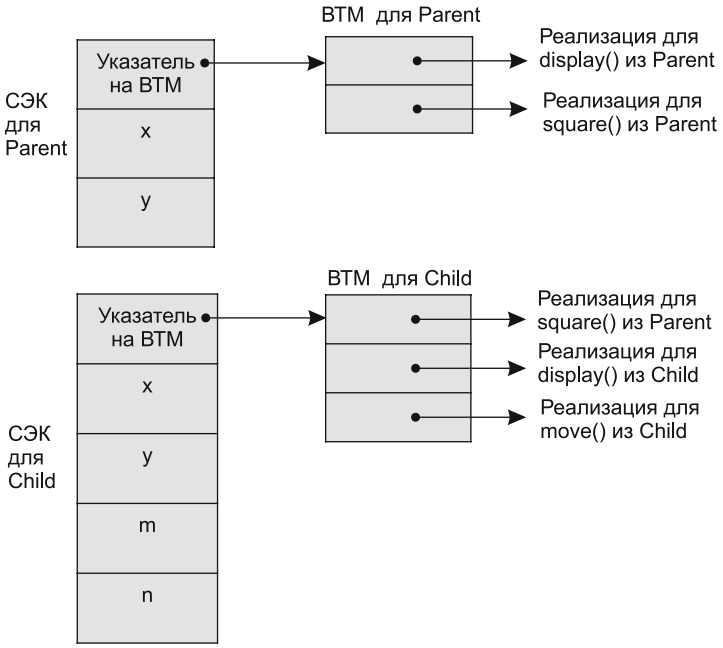


Рис. 16.24. Структуры СЭК и таблицы виртуальных методов для классов Parent и Child

Множественное наследование усложняет реализацию динамического связывания. Рассмотрим следующие определения классов на языке C++:

```
class A {
public:
    int a;
    virtual void opA1() { . . . }
    virtual void opA2() { . . . }
};
class B {
public:
    int b;
```

продолжение ⇩

```
    virtual void opB() { . . . }
};
class C : public A, public B {
public:
    int c;
    virtual void opA1() { . . . }
    virtual void opC() { . . . }
};
```

Классе C наследует переменную a и метод opA2() из класса A. Он переопределяет метод opA1(), хотя и сам метод, и его родительская версия потенциально видимы через полиморфную переменную (типа A). От класса B класс C наследует переменную b и метод opB(). Кроме того, класс C определяет свою собственную переменную c и добавляет не унаследованный метод opC(). Структура СЭК для C должна содержать данные классов A, B и C, а также средства для доступа ко всем видимым методам.

При единичном наследовании СЭК содержит указатель на виртуальную таблицу, которая адресует реализации всех видимых методов. При множественном наследовании все не так просто. В СЭК должны отражаться два представления: по одному для каждого родительского класса, причем в одно из них нужно включить представление для подкласса C. Подобное включение представления подкласса в представление родительского класса должно быть эквивалентно реализации единичного наследования.

Необходимы две виртуальные таблицы: одна для представления классов A и C, другая — для представления класса B. В этом случае первая часть СЭК для класса C содержит представление классов C и A со следующей структурой:

1) указатель виртуальной таблицы для методов C и тех методов, которые унаследованы от A;
2) данные, унаследованные от A.

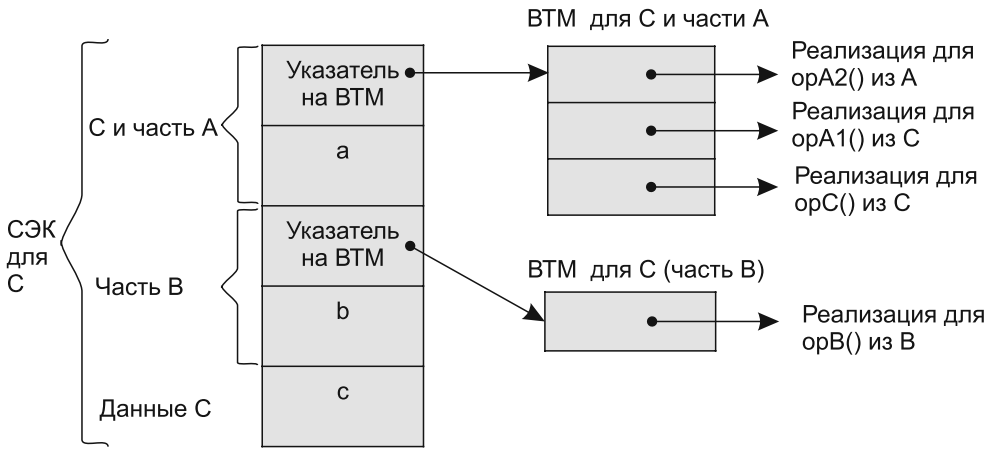


Рис. 16.25. Пример СЭК для подкласса с двумя родителями

Вторая часть в СЭК для класса C должна представлять класс B. Эта часть имеет следующую структуру:

1) указатель виртуальной таблицы для виртуальных методов B;
2) данные, унаследованные от B;
3) данные, определенные в C.

Структура СЭК для класса C показана на рис. 16.25.

Особенности аспектно-ориентированного подхода

Современные программные системы решают большое количество задач и характеризуются большой сложностью. Для преодоления этой сложности в ходе разработки используют проверенный принцип модульной декомпозиции. *Модульная декомпозиция* предполагает разделение систем на *иерархические* (то есть модульные) элементы (модули, компоненты, классы, объекты, функции, процедуры и т. д.). «Иерархичность» подразумевает, что элемент может состоять из других элементов, эти последние — еще из нескольких элементов и т. д. Границы этих элементов устанавливаются таким образом, что в них оказываются связанные друг с другом «соседи». Цель состоит в том, чтобы обеспечить тесную связь элементов в рамках каждого отдельного элемента и свести к минимуму взаимодействие между элементами.

Необходимость сосредоточения на решении какой-то одной проблемы Дейкстра называл принципом разделения задач или понятий (principle of separation of concerns) [11]. Являясь одним из фундаментальных принципов инженерии, разделение задач (понятий) широко применяется на всех этапах программной разработки.

Во всех языках программирования присутствуют конструкции, позволяющие структурировать описания систем в виде иерархических композиций мелких модульных элементов. Эти конструкции ориентированы на выявление и построение *функциональных компонентов*, которые выражаются в виде классов, объектов, модулей, процедур и т. д.

Недостаток функциональных решений состоит в том, что некоторые понятия (задачи) наподобие синхронизации, взаимодействия компонентов, устойчивости и управления защитой, которые имеют дело с *несколькими функциональными компонентами*, не поддаются четкой локализации. Как правило, они реализуются мелкими фрагментами кода, разбросанными среди множества функциональных компонентов.

Это последнее обстоятельство лежит в основе аспектно-ориентированного подхода (АОП) — нового направления в программировании [29, 42, 72, 87, 91]. Аспектно-ориентированное программирование предлагает в дополнение к *функциональным компонентам* использовать *аспекты*, которые «пересекают» функциональные компоненты, и предусматривает их композицию в целях получения качественных программных реализаций.

Примерами пересекающих задач (понятий) для аспектов являются синхронизация, ограничения реального времени, проверка ошибок, параллельное взаимодействие объектов, устойчивость, прослеживание истории, защита, политика кеширования, профилирование, мониторинг, тестирование и т. д. Некоторые системы характеризуются очень широким участием аспектов. К примеру, среди аспектов

распределенных систем числятся взаимодействие компонентов, синхронизация, удаленный вызов, стратегии передачи параметров, выравнивание нагрузки, репликация, нейтрализация сбоев, качество обслуживания и распределенные транзакции [42].

Одни аспекты соответствуют тем структурам, которые по природе своей проходят «поперек» обобщенных процедур, пересекают их — в частности, потокам управления и данных. К примеру, синхронизация, ограничения реального времени и взаимодействие объектов соответствуют потоку управления; стратегии передачи параметров в распространенных системах, напротив, соответствуют потоку данных.

Другая категория аспектов обеспечивает предметно-ориентированную оптимизацию. В ходе оптимизации выполняются прослаивание и клонирование — таким образом, одни подвергающиеся оптимизации высокоуровневые понятия перемежаются (сливаются), другие — распределяются среди низкоуровневых понятий. Эти явления составляют основу так называемого *спутывания и разбрасывания кода* оптимизированной версии — разобраться в нем и обеспечить его сопровождение становится достаточно проблематично. При попытке реализации аспектов средствами одних лишь обобщенных процедур спутывания кода не избежать.

Характерной чертой любых аспектов является пересечение: аспекты пересекают четкие иерархические системы (проходят их «поперек»), в рамках которых существуют модульные элементы декомпозиции. Этот принцип проиллюстрирован на рис. 16.26.

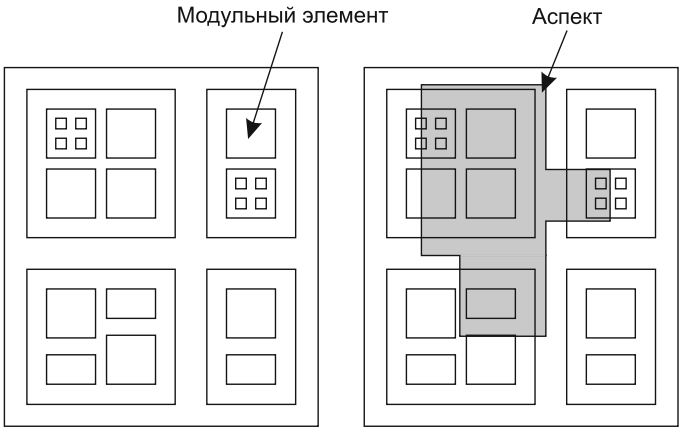


Рис. 16.26. Модульный и аспектный варианты декомпозиции

Согласно левой иллюстрации, модульные элементы четко разграничены и систематизированы в рамках иерархии. Иллюстрация справа говорит о том, что аспект может *пересекать* (crosscut) сразу несколько модульных элементов.

Практическое применение аспектов вводит понятие *точек соединения* (join points). Являясь элементами поведения, аспекты ссылаются в этих точках на выполнение пересекаемых объектов и других модулей.

Предположим, что у нас есть ограниченный буфер, а задача заключается в том, чтобы синхронизировать доступ к нему в многопоточной среде. Иначе говоря, клиенты, работающие в разных потоках, должны иметь возможность обращаться к буферу, помещать в него элементы и извлекать их. В силу того, что буфер является

совместно используемым ресурсом, клиентские обращения к нему должны быть синхронизированы.

В принципе, буфер и аспект синхронизации лучше всего реализовать в виде двух отдельных блоков кода (рис. 16.27). При этом аспект синхронизации должен задавать некие синхронизационные ограничения: «нельзя помещать и извлекать элементы одновременно», «нельзя помещать элементы в переполненный буфер», «нельзя извлекать элементы из пустого буфера» и т. д. Это пример классической «синхронизации ограниченного буфера».

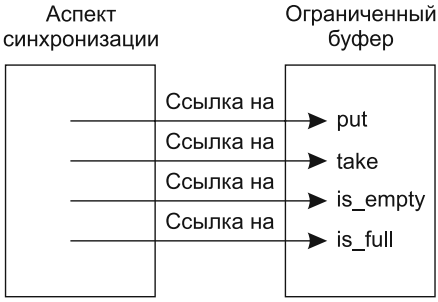


Рис. 16.27. Ограниченный буфер и его аспект синхронизации

Выскажем два существенных соображения:

1. Аспект «собирает некие сведения *о*» буфере. Для этого он обращается к элементам буфера. В нашем случае аспект синхронизации ссылается на методы буфера `put`, `take`, `is_empty` и `is_full`.
2. В отличие от буфера, который может использоваться «автономно» (например, в однопоточной программе), аспект синхронизации не обладает такой возможностью. Сам по себе он бесполезен. Свои функциональные возможности он проявляет исключительно в сочетании с конкретным буфером.

Теперь рассмотрим требования к взаимодействию аспекта и компонента (на примере буфера):

Минимизация сцепления. Сцеплением называют взаимодействие модулей друг с другом. Целесообразно,, чтобы количество ссылок на компонент в составе аспекта было минимальным. В данном примере аспект ссылается на методы `put()`, `take()`, `is_empty()` и `is_full()` буфера — не будь этих ссылок, сформулировать синхронизационные ограничения было бы невозможно. Для связывания аспекта синхронизации и компонента задействуют точки соединения.

Выбор времени и режимов связывания. Нужно сделать так, чтобы любой аспект можно было подключить или отключить в любой момент времени. При координации нескольких компонентов средствами синхронизации это требование приобретает особую важность. К примеру, если компоненты перенастраиваются динамическим способом, то подключение и отключение аспектов синхронизации также должно производиться динамически.

Неразрушающая адаптивность. Дополнение компонента аспектом синхронизации не должно быть связано с необходимостью ручного редактирования кода этого компонента. Возможность взаимодействия устанавливается путем динамического

или статического вплетения. *Вплетение* обеспечивает «вставку» кода аспекта в соответствующие участки компонента.

Зададимся вопросом: какие виды пересечения аспектов и модулей встречаются в объектно-ориентированных программах?

Во-первых, пересечение может существовать на уровнях классов и экземпляров. Если код какого-либо аспекта пересекает классы, то пересечение экземпляров этого класса регулируется именно на уровне классов. В нашем примере синхронизационный код пересекает буфер не только на уровне классов, но и на уровне экземпляров буфера. Пересечение на уровне классов предполагает существование механизмов организации взаимодействия, при помощи которых можно ассоциировать код аспектов с классами. Что касается состояния аспекта, то на уровне классов оно может либо быть общим для всех экземпляров, либо уникальным для каждого из них.

Пересечение на уровне экземпляров предполагает возможность ассоциирования кода аспектов с отдельными экземплярами. В частности, различные экземпляры структуры данных можно ассоциировать с кодом различных аспектов, реализующих те или иные стратегии синхронизации.

На рис. 16.28 различные виды пересечения классов и экземпляров сведены в единую схему. Каждое сочетание одного из элементов слева с одним из элементов справа представляет отдельный вид пересечения. Обратим внимание: пересечение отдельного аспекта с отдельным классом/объектом происходит на уровне методов.

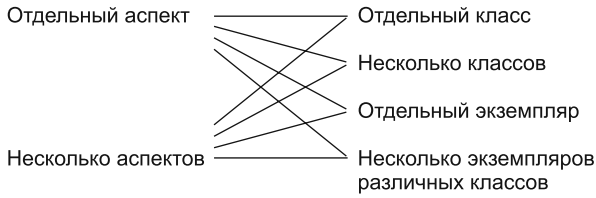


Рис. 16.28. Различные виды пересечения классов и экземпляров

Точки соединения сообщений в объектно-ориентированном программировании встречаются довольно часто. Применительно к аспектам список точек соединения должен быть продолжен: обеспечение параллелизма, ограничения реального времени, элементарные транзакции, проверка ошибок по предусловиям, защита, профилирование, мониторинг, тестирование, кеширование и прослеживание истории. Как правило, соответствующие аспекты реализуются в виде объектов и при помощи определенных механизмов сцепления сообщений привязываются к объектам базовой структуры. На уровне методов существует широкий спектр возможностей — реализацию метода или отдельный его вызов можно дополнить предварительной операцией, вторичной операцией или и тем, и другим. Что касается вызова методов, то здесь можно встраивать операции внутрь реализаций методов — для этого внутри одного метода записываются вызовы других методов.

Мы уже отмечали, что на уровне системы добавление кода реализации пересечений приводит к двум нежелательным явлениям: спутыванию и разбрасыванию.

Спутывание происходит, когда методы класса включает в себя дополнительный код, который реализует различные требования к системе. Например, во все методы

класса Студент добавляются вызовы методов безопасность( ) и восстановление( ), реализующих понятия безопасности и восстановления.

Явление *разбрасывания* возникает при реализации такого понятия (задачи), которое разбрасывается по многим классам системы.

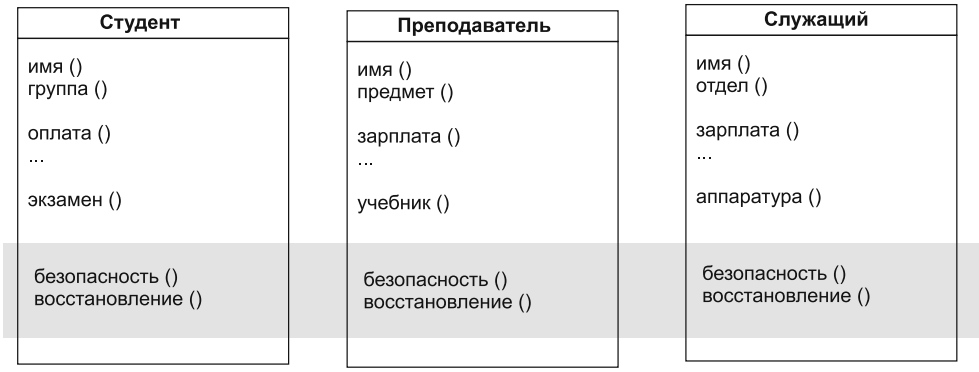


Рис. 16.29. Разбрасывание кода

На рис 16.29 приведен пример трех классов для информационной системы института. Затененная область показывает методы, необходимые для реализации пересекающих понятий безопасности и восстановления. Легко видеть, что реализация этих понятий разбросана по реализациям всех основных понятий системы.

Проблемы с разбрасыванием и спутыванием часто происходят при добавлении новых требований к системе. Представьте, что появляется новое требование сбора статистики по всем сотрудникам института. Эта информация находится в разных местах системы, и поэтому потребуется много времени на поиск изменяемых классов, внесение и проверку изменений. Причем всегда существует возможность пропуска некоторого кода. Кроме того, чем больше изменений необходимо сделать, тем больше вероятность внести ошибки в систему.

Базовые понятия АОП

Для поддержки механизма пересечения система аспектно-ориентированного программирования должна предоставлять следующие строительные элементы.

Идентифицируемые точки в вычислительном процессе программы. В ходе вычислений можно выделить характерные точки. Они могут отмечать вызов методов, их выполнение, создание объектов или формирование исключений. Такие опознаваемые точки в вычислительном процессе называют *точками соединения* (*join points*). Точки соединения присутствуют в любых вычислениях, поскольку просто отражают определенные ситуации в процессе выполнения программы. АОП лишь идентифицирует и классифицирует эти точки.

Механизм для выбора точек соединения. Реализация пересекающего понятия требует выбора конкретных точек соединения. Например, аспект трассировки может потребовать распознавания только публичных методов. Элемент «*срез*» (*pointcut*) выбирает любую точку соединения, удовлетворяющую этому критерию. Срез

может использовать другой срез, чтобы обеспечить сложный выбор. Срезы также оценивают контекст в выбранных точках. Например, срез может считать контекстом аргументы некоторого метода. Концепция точек соединения и логическая структура среза совместно формируют *модель точек соединения* системы АОП.

Составной оператор для изменения поведения программы. После того как срез выберет точки соединения, для этих точек следует определить дополнительное или альтернативное поведение. Оператор *«совет»* (*advice*) является средством реализации такого поведения. Совет добавляет поведение перед, после или вокруг отобранных точек соединения. Совет «перед» будет выполняться перед точкой соединения, а совет «после» — после этой точки. Совет «вокруг» окружает вычисления в точке соединения и может выполняться от нуля до множества раз. Совет — это форма *динамичного пересечения*, потому что воздействует на вычисления программы.

Объявления для изменения статической структуры программы. Иногда для реализации пересекающей функциональности необходимо изменить статическую структуру программы. Например, при реализации трассировки вам потребуется ввод поля регистрации в каждый трассируемый класс; *межтиповые объявления* (*inter-type declaration*) делают такие модификации возможными. В некоторых ситуациях может понадобиться выявление определенных условий (к примеру, проверка наличия специфических точек соединения перед выполнением программы); подобный анализ обеспечивает *объявление времени вплетения* (*weave-time declaration*). Коллективно все эти средства формируют механизм *статического пересечения*, поскольку воздействуют на статическую структуру, в противоположность изменениям динамичного поведения в выполнении программы.

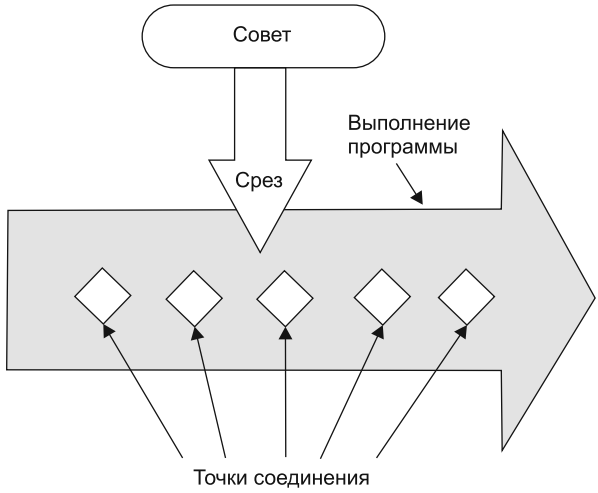


Рис. 16.30. Функциональная часть аспекта (совет) вплетается в процесс выполнения программы в одной или нескольких точках соединения

Синтаксическая капсула для размещения всех средств пересечения. Поскольку конечная цель АОП — иметь конструкцию, которая содержит все средства пересечения, необходим специальный синтаксический контейнер. Подобным контейнером является конструкция *«аспект»* (*aspect*). Конструкция «аспект» несет в себе срезы,

совет и объявления статического пересечения. Она может быть связана с другими аспектами, подобно связи одного класса с другими классами. Аспекты становятся частью программы и используют ее (например, классы программы), чтобы реализовать пересечения. Динамическую природу аспекта иллюстрирует рис. 16.30.

Каждая конкретная система АОП может поддерживать лишь часть рассмотренных строительных элементов. Например, в АОП Spring не реализованы объявления времени вплетения. С другой стороны, модель точек соединения настолько важна, что любая система АОП должна поддерживать ее — все остальное является производным от модели точек соединения.

Аспекты

При пояснении аспектно-ориентированного программирования воспользуемся терминологией самого популярного языка в этой области — AspectJ. Основные термины представлены в табл. 16.5.

Таблица 16.5. Основные термины аспектно-ориентированного подхода

Термин	Пояснение
Advice — совет	Код, реализующий пересекающее понятие
Aspect — аспект	Программная конструкция, которая определяет пересекающее понятие. Она включает в себя определение среза и советы, связанные с этим понятием
Join point — точка соединения	Событие в процессе выполнения программы, когда соответствующий аспекту совет может быть выполнен
Join point model — модель точек соединения	Набор событий, на которые ссылаются в срезе
Pointcut — срез	Инструкция, включаемая в аспект, которая определяет точки соединения, где соответствующий совет должен быть выполнен
Weaving — вплетение	Включение ткачом кода совета в указанной точке соединения

Жизнь без аспектов

Как вы реализуете пересекающие понятия, пользуясь только ООП? Обычно вы добавляете код, нужный для каждого пересекающего понятия, в каждый класс. В итоге каждый класс в программе реализуют как основные, так и пересекающие понятия. Проиллюстрируем эту идею посредством фрагмента кода. Рассмотрим структурную реализацию класса, который инкапсулирует несколько понятий традиционным способом (программа состоит из многих таких классов:

```
public class SomeBusinessLogic {
    Основные элементы данных
    Элемент данных для регистрации // Трассировка
    Блокировка управления параллелизмом // Контроль распараллеливания
```

продолжение ⇩

```
    public void operation1(<parameters>) {
        Проверить авторизацию // Проверка безопасности
        Блокировка для обеспечения сохранности потока // Контроль распараллеливания
        Начало транзакции // Управление транзакцией
        Регистрация начала операции // Трассировка
        Выполнение самой операции
        Регистрация завершения операции // Трассировка
        Завершение или откат транзакции // Управление транзакцией
        Разблокировка для обеспечения сохранности потока // Контроль
                                                          // распараллеливания}
    ... Другие операции, реализующие множество понятий
}
```

Хотя детали могут меняться, фрагмент иллюстрирует общую проблему, с которой сталкиваются разработчики: на уровне архитектуры программы различные понятия концептуально разделяются, однако реализация сплетает их, делая класс ответственным за собственное понятие и многие пересекающие понятия. Если вам нужно изменить вызов кода, связанного с пересекающим понятием, вы должны изменить каждый класс, который включает такой вызов. При традиционной реализации основное и пересекающие понятия спутываются в каждом классе. К тому же каждое пересекающее понятие разбросано по многим классам.

Жизнь с аспектами

В ООП основные понятия достаточно слабо связаны через интерфейсы, чего не скажешь о пересекающих понятиях. Дело в том, что понятие реализуется в двух частях: серверной части и клиентской части. С помощью классов и интерфейсов в ООП серверная часть легко разбивается на модули. Но когда понятие имеет пересекающую природу, клиентская часть (состоящая из запросов к серверу) распространяется по всем клиентам.

ПРИМЕЧАНИЕ

Термины «сервер» и «клиент» используются здесь в классическом смысле, принятом в ООП, что предполагает объекты, которые обеспечивают определенный набор услуг, и объекты, пользующиеся этими услугами.

В качестве иллюстрации рассмотрим реализацию пересекающего понятия безопасности в ООП. Экземпляр класса безопасности предоставляет свои услуги через интерфейс. Использование интерфейса ослабляет связанность между клиентами и серверами (реализациями интерфейса). Клиенты, которые пользуются услугами безопасности через интерфейс, не замечают фактической реализации того, чем они пользуются; никакие изменения в реализации не требуют изменений непосредственно в клиентах. Аналогично, замена одной реализации безопасности на другую — это только способ подтверждения правильности способа реализации. Как результат, вы можете заменить одну реализацию безопасности другой без изменений в индивидуальных объектах-клиентах. Но эта расстановка все еще требует, чтобы каждый клиент имел встроенный код для вызова объекта безопасности. Такие вызовы должны быть включены во все объекты, требующие безопасности, и спутываются с их базовой логикой.

При использовании АОП ни один из основных объектов не содержит вызова объекта безопасности. Понятие безопасности — реализация плюс вызовы — теперь полностью и постоянно располагается внутри объекта безопасности и аспекта безопасности.

Фундаментальное изменение, которое привносит АОП — это сохранение взаимной независимости индивидуальных понятий. Реализации могут легко отображаться обратно на соответствующие понятия, поэтому программа упрощается и легко адаптируется к изменениям.

Правила вплетения

Гибкость языка АОП во многом определяется принятыми в нем правилами вплетения аспектов. Правила вплетения конкретизируют, как комбинировать классы, объекты и аспекты, чтобы сформировать окончательную программу.

Мощь языка АОП проистекает из экономного способа выражения правил вплетения. Например, для выделения из класса `SomeBusinessLogic` (рассмотренного в предыдущем подразделе) понятия трассировки, достаточно добавить несколько строк кода, конкретизирующих, что все публичные операции в программе требуют регистрации. Вот спецификация вплетения для аспекта трассировки:

- Правило 1. Создать объект регистратора.
- Правило 2. Регистрировать начало каждой публичной операции.
- Правило 3. Регистрировать завершение каждой публичной операции.

Это гораздо проще, чем добавление в каждую публичную операцию кода для регистрации. После применения правил и к другим пересекающим понятиям в классе останется только основная бизнес-логика:

```
public class SomeBusinessLogic {
    Основные элементы данных
    public void operation1(<parameters>) {
    Выполнение самой операции
    ... Другие операции
}
```

Сравните этот вариант с исходным классом — вспомогательные (с точки зрения класса) пересекающие понятия удалены, они ушли в аспекты.

Правила вплетения могут быть общими или специфическими в зависимости от того, как аспекты взаимодействуют с основными объектами. В рассмотренном примере правила вплетения не нуждаются в упоминании каких-либо классов и методов программы. В другом случае правило вплетения может конкретизировать, что аспект должен применяться только к специфическим методам, например `credit()` и `debit()` в классе `Account` и т. д. Специфичность правил вплетения определяет уровень сцепления между аспектом и базовой логикой.

Применение аспектов

Вообразим, что в информационной системе института произошло нарушение безопасности, из-за которого изменился размер зарплаты какого-то преподавателя. Может быть, кто-то случайно оставил компьютер подключенным, и неизвестный человек получил несанкционированный доступ к системе. С другой стороны, служащий, имеющий доступ к системе, может преднамеренно поменять чью-то зарплату.

Для исключения подобной возможности вводится новая стратегия безопасности. Перед любым изменением в базе данных пользователь, производящий изменения, должен заново идентифицироваться в системе. Подробная информация о том, кто сделал изменения, также регистрируется в отдельном файле.

Как реализовать эту стратегию?

- *Первый путь.* Изменить содержание метода обновления данных в каждом классе (добавить туда вызов метода проверки подлинности и метода журналирования). Это приводит к спутыванию кода. Логически обновление базы данных, проверка подлинности пользователя и журналирование события являются отдельными, не связанными понятиями. Кому-то захочется проверить подлинность где-то еще без журналирования. Третье лицо пожелает записывать события в журнал без обновления информации в БД.
- *Второй путь.* Можно поступить иначе: перед каждым вызовом метода обновления информации выполнять вызовы проверки подлинности и журналирования, добавив эти методы в каждый класс. Здесь мы получаем разбросанную реализацию.

Лучшим решением этой проблемы является применение аспекта:

```
aspect проверкаПодлинности {
    before: call (public void обновить*(..)) // это срез
    // этот совет вплетается в процесс выполнения
    int попытка = 0;
    string userPassword = Password.Get(попытка);
    while (попытка < 3 && userPassword != thisUser.password()){
        // разрешаются 3 попытки ввода пароля
        попытка = попытка + 1;
        userPassword = Password.Get(попытка); {
    }
    if (userPassword != thisUser.password()) then
        // при неправильном пароле выход из системы
        System.Logout(thisUser.uid);
    public void включить() {
    }    ...

} // проверкаПодлинности
```

ПРИМЕЧАНИЕ

Мы использовали синтаксис, близкий к синтаксису языка AspectJ, но сознательно упростили его.

Аспекты полностью отличаются от других программных конструкций тем, что аспект включает в себя определение места, где он должен быть выполнен. В других конструкциях, например в методах, существует четкое разделение между определением абстракции и ее использованием. Изучив метод, нельзя сказать, в каком месте он будет вызван. Вызовы могут быть выполнены абсолютно из любого места в программе. Аспекты, напротив, включают *определение среза* (pointcut), утверждения, задающего то место, где аспект будет вплетен в процесс выполнения.

В данном примере срез является простой инструкцией:

```
before: call (public void обновить*(..))
```

Смысл инструкции в том, что перед выполнением любого метода, имя которого начинается со слова `обновить`, за которым следует любая другая последовательность символов, должен быть выполнен код в аспекте, который следует после определения среза. Символ звездочка (*) соответствует любой строке символов, разрешенных в идентификаторах. Код, который должен выполняться, известен как «совет» и является реализацией пересекающего понятия. В данном примере совет получает пароль от человека с просьбой об изменении и проверяет его совпадение с паролем пользователя, вошедшего в систему. Если совпадения нет, пользователь выводится из системы и обновление не происходит.

Возможность указывать с помощью среза, где код должен быть выполнен, является отличительной чертой аспектов. Однако чтобы понять, что такое срез, следует разъяснить другую концепцию — идею точек соединения. Точки соединения являются событиями, которые происходят во время выполнения программы: это может быть вызов метода, инициализация переменной, обновление атрибута и т. д.

Существует множество типов событий, происходящих во время выполнения программы. Модель точек соединения определяет набор событий, на которые можно ссылаться в аспектно-ориентированных программах. Модель точек соединения не стандартизована, и каждый аспектно-ориентированный язык программирования предлагает свою модель точек соединения. Например, модель точек соединения языка AspectJ включает в себя:

- События вызова — вызовы метода или конструктора.
- События выполнения — выполнение метода или конструктора.
- События инициализации — инициализация объектов или классов.
- События данных — доступ к атрибуту или модификация атрибута.
- События исключений — обработка исключений.

Срез определяет конкретное событие (события) (например, вызов именованной процедуры), с которым ассоциируется совет. Это означает, что советы можно вплетать в программу в различных случаях, определяемых моделью точки соединения:

- Совет может быть включен до выполнения конкретного метода, списка конкретных методов или списка методов, имена которых соответствуют шаблону (например, `обновить*`).
- Совет может быть включен после обычного или необычного (по исключению) возврата из метода. Например, точка среза `after : returning (public void обновить * (..))` может задавать журналирование после каждого выполнения любого метода обновления.
- Совет может быть включен, когда происходит доступ к атрибуту объекта. Вы можете внедрить совет для трассировки или изменения значения этого атрибута.

Включение совета в точках соединения, описанных в срезе, — задача ткача аспектов. Ткач аспектов — это расширение компиляторов, которое обрабатывает описания аспектов, а также объекты классов и методы, составляющие систему. Далее ткач создает новую программу, в которой аспекты включены в описанных точках соединения. Аспекты интегрируются в систему так, что пересекающие понятия выполняются в нужных местах конечной системы.

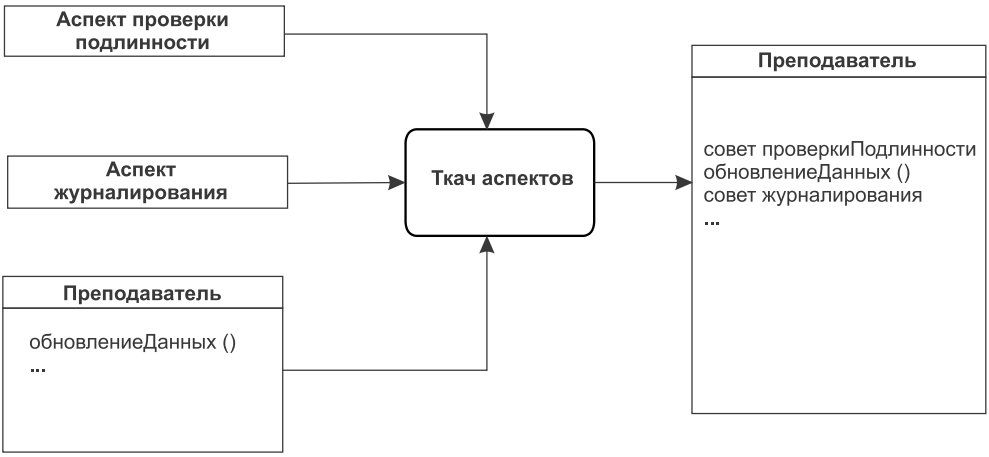


Рис. 16.31. Вплетение аспектов

На рис. 16.31 показано вплетение двух аспектов для реализации проверки подлинности и журналирования в информационной системе института.

Существует три разных подхода к вплетению аспекта:

1. Предварительная обработка исходного кода, где ткач из исходного кода генерирует новый исходный код на таких языках, как Java или C++, который может быть далее скомпилирован стандартным компилятором соответствующего языка.
2. Вплетение во время компоновки, когда компилятор модифицируется для включения в него ткача. В данном случае обрабатывается текст на аспектно-ориентированном языке, таком как AspectJ, и генерируется стандартный байт-код Java. Конечный результат может исполняться непосредственно интерпретатором Java или перерабатываться в машинный код.
3. Динамическое вплетение во время выполнения. В этом случае наблюдаются точки соединения, и когда происходит событие, описанное в срезе, соответствующий совет интегрируется в исполняемую программу.

Наиболее часто используемый подход к вплетению аспектов — это вплетение во время компоновки, так как обеспечивает эффективную реализацию аспектов без больших накладных расходов во время выполнения. Динамическое вплетение — наиболее гибкий подход, но может значительно ухудшить производительность программы. Предварительная обработка исходного текста в настоящее время используется редко.

Программирование на аспектно-ориентированном языке AspectJ

Рассмотрим простой класс на языке Java, содержащий два метода, которые печатают сообщения. Позднее мы создадим несколько аспектов, расширяющих его поведение без модификации класса:

```
package ajia.messaging;
public class MessageCommunicator {
    public void deliver(String message) {
        System.out.println(message);
    }
    public void deliver(String person, String message) {
        System.out.println(person + ", " + message);
    }
}
```

Класс `MessageCommunicator` имеет два метода: один для доставки общего сообщения, а другой для доставки сообщения конкретному человеку

Рассмотрим функциональность аутентификации: перед доставкой сообщения вы хотели ли бы проверить, был ли аутентифицирован пользователь. Без использования АОП пришлось бы написать следующий код:

```
public class MessageCommunicator {
    private Authenticator authenticator = new Authenticator();
    public void deliver(String message) {
        authenticator.authenticate();
        System.out.println(message);
    }
    public void deliver(String person, String message) {
        authenticator.authenticate();
        System.out.println(person + ", " + message);
    }
}
```

Вам пришлось добавить код вызова `authenticate()` в каждый метод, которому нужна аутентификация, что ведет к спутыванию кода. Подобный код должен присутствовать во всех классах, где требуется функциональность аутентификации, что ведет к разбрасыванию кода. АОП предлагает лучшее решение.

Без изменения содержания класса `MessageCommunicator` его функциональность можно увеличить добавлением к программе аспекта:

```
package ajia.security;
import ajia.messaging.MessageCommunicator;
public aspect SecurityAspect { // Объявление аспекта
    private Authenticator authenticator = new Authenticator();
    pointcut secureAccess() // Объявление среза
    : execution(* MessageCommunicator.deliver(..));
    before() : secureAccess() { // Совет
    System.out.println("Checking and authenticating user");
    authenticator.authenticate();
    }
}
```

Класс `Authenticator` запрашивает полномочия (имя пользователя и пароль), когда в потоке метод `authenticate()` вызывается первый раз. При успешной аутентификации класс сохраняет пользователя в локальной области потока; таким образом, в этом потоке полномочия повторно не запрашиваются. При ошибке генерируется исключение периода выполнения.

Рассмотрим содержание аспекта более детально:

1. Аспект — это основной модуль в АОП, подобно тому, как класс — основной модуль в ООП. Объявление аспекта подобно объявлению класса.

2. Срез выделяет в процессе выполнения программы необходимые точки, называемые точками соединения. Аспект определяет срез `secureAccess()`, который выделяет процессы выполнения всех методов с именем `deliver()` из класса `MessageCommunicator`. Символ * указывает, что срез сопоставляется с любым возвращаемым типом, а две точки внутри круглых скобок (..) после `deliver` определяют, что срез сопоставляется с любым количеством и типами параметров. В этом примере срез выбирает процессы выполнения обеих перегруженных версий `deliver()` в классе `MessageCommunicator`.
3. Совет определяет код, который должен быть выполнен при достижении точек соединения, выделенных соответствующим срезом. Здесь записан совет, который должен выполняться перед достижением точек соединения, выбранных срезом `secureAccess()`. Предложение `before()` указывает, что совет должен быть запущен прежде выполнения точки соединения, определяемой советом, — в данном случае прежде выполнения любого метода `MessageCommunicator.deliver()`. Цель данного совета: провести аутентификацию текущего пользователя. Присутствие в программе этого аспекта гарантирует, что перед каждым выполнением метода `MessageCommunicator.deliver()` код совета реализует логику аутентификации.

   Перейдем к обзору основных строительных блоков языка AspectJ.

Конструкции пересечения языка AspectJ

Язык AspectJ является наиболее полной реализаций рассмотренных нами понятий АОП. Следует выделить три группы конструкций пересечения: общие конструкции пересечения (точка соединения, срез и аспект), конструкции динамического пересечения (совет) и конструкции статического пересечения (межтиповые объявления `inter-type` и объявления времени вплетения `weave-time`).

Общие конструкции пересечения

Общие конструкции пересечения можно использовать как при динамическом, так и при статическом пересечении.

В AspectJ точки соединения определяют позиции в вычислительных процессах, куда внедряются пересекающие действия.

После идентификации возможных точек соединения, полезных для пересекающей функциональности, нужно выбрать из них необходимые точки, используя конструкцию среза.

Срез решает задачу выбора необходимых точек и сбора информации о контексте в этих точках. Например, срез может выбрать точку соединения, которая соответствует процессу выполнения метода. Он способен выделить такой контекст точки соединения, как объект `this` и параметры метода.

Следующий срез выделяет процессы выполнения любого публичного метода в программе:

```
execution(public * *.*(..))
```

Символы подстановки * и (..) здесь указывают, что срез делает выбор независимо от возвращаемого типа, объявленного класса (объекта), имени и параметров ме-

тода. В расчет принимается единственное условие: спецификатор доступа к методу должен быть задан как `public`.

Рекомендуется давать срезу такое имя, которое могли бы использовать другие элементы программы (и чтобы программисты могли понимать смысл среза). Например, ранее упомянутый срез можно назвать `publicOperation`:

```
pointcut publicOperation() : execution(public * *.*(..));
```

Аспект является центральной единицей в AspectJ в той же степени, в которой класс считается центральной единицей в Java. Он содержит код, выражающий правила вплетения как для динамического, так и для статического пересечения. Дополнительно аспекты могут содержать данные, методы и элементы вложенного класса, подобно обычному классу Java. К примеру, определим аспект, который выполняет профилирование:

```
package ajia.profile;
public aspect ProfilingAspect {
}
```

Знания общих конструкций пересечения окупаются, когда вы начинаете их использовать для динамического и статического пересечения.

Конструкция динамического пересечения — совет

Поддержка динамического пересечения AspectJ представлена в форме совета. Совет — это составной оператор, выполняемый в точке соединения, выбранной срезом. Совет может выполняться перед, после или вокруг точки соединения. Тело совета во многом подобно телу метода — оно задает действия, которые должны быть выполнены по достижении точки соединения.

Пользуясь срезом `publicOperation()` из предыдущего подраздела, можно организовать следующее профилирование всех публичных методов из класса `MessageCommunicator`:

```
package ajia.profile;
public aspect ProfilingAspect {
    pointcut publicOperation() : execution(public * *.*(..));
    Object around() : publicOperation() {
        long start = System.nanoTime();
        Object ret = proceed();
        long end = System.nanoTime();
        System.out.println(thisJoinPointStaticPart.getSignature()
            + " took " + (end-start) + " nanoseconds");
        return ret;
    }
}
```

Совет записывает время старта, вызывает `proceed()` для продолжения выполнения «советуемого» метода, фиксирует время завершения и печатает длительность выполнения метода. Переменная `thisJoinPointStaticPart` — одна из трех переменных, доступных в каждом совете, которая несет такую информацию о «присоветованной» текущей точке соединения, как имя метода, объект `this` и параметры метода.

В то время как динамическое пересечение изменяет поведение программы, статическое пересечение изменяет структуру программ.

Конструкции статического пересечения

Межтиповое объявление, называемое также *введением*, является конструкцией статического пересечения, которая изменяет статическую структуру классов, интерфейсов и аспектов программы. К примеру, в класс можно добавить метод или поле или объявить тип для реализации интерфейса. В этом объявлении один тип (аспект) объявляет структуру других типов (классов, интерфейсов и даже аспектов) — отсюда и название.

Следующий оператор предлагает ткачу назначить интерфейс `AccessTracked` родителем класса `MessageCommunicato`:

```
declare parents: MessageCommunicator implements AccessTracked;
```

Когда это объявление будет вплетено, эффект будет равносилен такому объявлению класса `MessageCommunicator`:

```
public class MessageCommunicator implements AccessTracked {
    ...
}
```

Другая форма межтипового объявления — введение элемента — предлагает путь добавления новых методов и полей в другие типы. Следующее объявление добавляет поле `lastAccessedTime` и методы `updateLastAccessedTime()` и `getLastAccessedTime()` к типу `AccessTracked`:

```
private long AccessTracked.lastAccessedTime;
public void AccessTracked.updateLastAccessedTime() {
    lastAccessedTime = System.currentTimeMillis();
}
public long AccessTracked.getLastAccessedTime() {
    return lastAccessedTime;
}
```

Далее можно посоветовать методам в типе, который реализует `AccessTracked` (непосредственно или через оператор `declare parents`), обновить время последнего доступа, как показано в следующем фрагменте:

```
before(AccessTracked accessTracked)
    : execution(* AccessTracked+.*(..))
    && !execution(* AccessTracked.*(..))
    && this(accessTracked){
    accessTracked.updateLastAccessedTime();
}
```

Этот код советует всем методам типов, которые реализуют интерфейс `AccessTracked` (символ подстановки + обозначает подтипы), но не методу в `AccessTracked` (например, введенному методу `updateLastAccessedTime()`). Срез `this()` выделяет трассируемый объект, следовательно, для него можно вызвать метод `updateLastAccessedTime()`.

Теперь разместим все эти фрагменты в аспекте:

```
package ajia.track;
import ajia.messaging.MessageCommunicator;
public aspect TrackingAspect {
    declare parents: MessageCommunicator implements AccessTracked;
    private long AccessTracked.lastAccessedTime;
    public void AccessTracked.updateLastAccessedTime() {
```

```
        lastAccessedTime = System.currentTimeMillis();
    }
    public long AccessTracked.getLastAccessedTime() {
        return lastAccessedTime;
    }
    before(AccessTracked accessTracked)
    : execution(* AccessTracked+.*(..))
        && !execution(* AccessTracked.*(..))
        && this(accessTracked) {
            accessTracked.updateLastAccessedTime();
        }
        private static interface AccessTracked {
    }
}
```

Следующая форма статического пересечения позволяет в ходе компиляции обнаружить и отметить точки соединения, которые соответствуют срезу.

Объявление времени вплетения является еще одной формой конструкции статического пересечения, позволяющей добавлять предупреждения и сообщения об ошибках времени вплетения при обнаружении определенных паттернов использования. Достаточно часто вплетение выполняется в ходе компиляции; поэтому эти предупреждения и сообщения об ошибках выдаются при компиляции классов.

Вернемся к аспекту `SecurityAspect`. Допустим, что следует получать предупреждения о непосредственных вызовах метода `Authenticator.authenticate()`. Следующее объявление принуждает формировать предупреждения о том, что любая часть программы вызывает запрещенный метод, исключением, конечно, остается аспект `SecurityAspect`:

```
declare warning
    : call(void Authenticator.authenticate()) && !within(SecurityAspect)
    : "Аутентификация должна выполняться только средствами SecurityAspect";
```

Срез `call()` выделяет вызов метода, а `!within()` ограничивает выбор теми точками соединения, которые появляются вне `SecurityAspect`. При обнаружении заданных условий ткач будет формировать предупреждения.

Можно добавить это объявление к аспекту SecurityAspect:

```
package ajia.security;
public aspect SecurityAspect {
    ...
    declare warning
    : call(void Authenticator.authenticate())
    && !within(SecurityAspect)
    : "Аутентификация должна выполняться только средствами SecurityAspect ";
}
```

Альтернативный синтаксис @AspectJ

Альтернативный синтаксис имеет версия @AspectJ, которая расширяет язык с помощью средства аннотации языка Java версии 5. Главное преимущество этого синтаксиса состоит в том, что код можно компилировать с помощью обычного компилятора Java. Среда Spring также понимает этот синтаксис. Недостатком

синтаксиса @AspectJ считается его «многословие» при записи тех же конструкций и ограниченные возможности представления некоторых конструкций, особенно в категории статического пересечения.

Перепишем средствами этого синтаксиса аспект `SecurityAspect`:

```
package ajia.security;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
    public class SecurityAspect { // Объявление аспекта
        private Authenticator authenticator = new Authenticator();
        @Pointcut(  // Срез
            "execution(* ajia.messaging.MessageCommunicator.deliver(..))")
        public void secureAccess() {}
        @Before("secureAccess()")  // Совет
        public void secure() {
            System.out.println("Checking and authenticating user");
            authenticator.authenticate();
        }
    }
```

Обсудим различия:

1. Вместо ключевого слова `aspect` используется слово `class` с аннотацией `@Aspect`. Компилятор Java, который понимает семантику этой аннотации, считает аннотированный класс аспектом.
2. Аналогично аннотация `@Pointcut` помечает метод как срез. Выражения среза записывается в стиле традиционного синтаксиса (за исключением того, что требуется указывать полностью квалифицированное имя для `ajia.messaging.MessageCommunicator`). Имя метода рассматривается как имя среза.
3. Аннотация `@Before` помечает обычный метод как совет `before`. Тело метода образует тело совета — оно выполняется перед выбранной точкой соединения.

Контрольные вопросы и упражнения

1. В объектно-ориентированной иерархии наследования каждый следующий уровень является более специализированной формой предыдущего. Приведите пример иерархии из повседневной жизни с этим свойством. Некоторые из иерархий, обнаруживаемые в реальной жизни, не являются иерархиями наследования. Укажите пример иерархии без свойства наследования.
2. Сравните инкапсуляцию, обеспечиваемую механизмом классов, и инкапсуляцию в императивных модулях. Чем они отличаются? В чем похожи?
3. Обоснуйте утверждение: если не используется ни замещение, ни уточнение, то подкласс всегда должен быть подтипом.
4. Приведите пример, иллюстрирующий, что если имеет место семантика замещения, то подкласс не обязан быть подтипом.
5. Хотя систематическое использование семантики уточнения делает более сложным создание подклассов, не являющихся подтипами, такое все же возможно.

Проиллюстрируйте это, приведя пример уточняющего подкласса, не являющегося тем не менее подтипом базового класса.

6. Предположим, нужно написать программный проект на языке программирования, который не является объектно-ориентированным (например, Pascal или C). Как бы вы имитировали классы и методы? Как бы вы имитировали наследование? Сможете ли вы обеспечить множественное наследование? Обоснуйте свой ответ.
7. Есть различные способы реализации структуры данных стек — например, с помощью списков или в виде массива. Предположим, мы имеем и класс списков List, и класс массивов Array. Для каждого из них проиллюстрируйте, как можно построить стек, используя и наследование, и композицию. Вам разрешается ввести какие угодно методы для базовых классов. Какая из техник реализации кажется вам более подходящей в данном случае?
8. Проиллюстрируйте, почему в объектно-ориентированных языках программирования со статическими типами данных (например, C++ или Delphi Pascal) компилятор проверяет, чтобы значение переменной родительского класса не присваивалось переменной, описанной как экземпляр подкласса.
9. Объясните, почему при условии соблюдения принципа подстановки возвращаемое значение для метода дочернего класса не может быть более общим, чем у родительского класса.
10. Приведите два примера множественного наследования в ситуациях, не связанных с компьютерами.
11. Известны случаи, когда практический пример множественного наследования в C++ поясняют введением класса IntegerArray, который наследует от двух классов Array и Integer. Как вы думаете, является ли это хорошим примером множественного наследования? Обоснуйте свой ответ.
12. Создайте класс Tree так, чтобы он мог быть использован как двоичное дерево. Обеспечьте средства для поиска или изменения левого или правого потомка любого узла. Какие предположения вам требуются?
13. Обсудите виртуальное наследование в языке C++ с точки зрения принципа сокрытия информации.
14. Какие операции обычно являются перегруженными в традиционных языках программирования (Pascal и C)?
15. Контейнерные классы в объектно-ориентированном программировании: успех или неудача?
16. Структуры данных разделяются на те, которые характеризуются своей реализацией (связные списки, деревья), и на те, которые определяются их предназначением (стеки, множества). Опишите, как можно использовать ООП для упрощения второго типа структур и сокрытия деталей их реализации. Приведите в качестве иллюстрации структуру данных с единым интерфейсом и двумя радикально отличающимися реализациями.
17. Защищенные данные (для клиентов-подклассов) занимают промежуточное положение по степени закрытости между открытыми и закрытыми полями. Представьте себе аналогичную промежуточную ступень между следующими

положениями: «доступ к объекту разрешен кому угодно» и «доступ к объекту осуществляет только сам объект». Например: «экземпляры некоторого класса (и только они) могут иметь доступ ко внутреннему состоянию объекта». В языке C++, например, экземпляр класса имеет доступ к любому полю данных другого объекта того же класса, даже если эти поля описаны как private или protected. В других языках (например, в Smalltalk) такое не разрешается. Обсудите преимущества и недостатки каждого из подходов.

18. Объясните, почему методы, которые в языке C++ не описаны как виртуальные, могут вызываться более эффективно, чем виртуальные методы. Как бы вы произвели замеры времени, чтобы определить, является ли разница существенной?
19. На хорошо знакомом вам языке программирования разработайте набор полиморфных функций (например, push, pop, , top), которые позволяли бы создавать стеки, содержащие элементы любого типа.
20. Зачем нужны чистые виртуальные классы в C++? Мы могли бы воздержаться от распределения в памяти объектов этого класса и просто использовать подклассы суперкласса.
21. Рассмотрим следующие объекты: прямоугольник, круг, квадрат, треугольник, многоугольник, прямая, точка, объект, четырехугольник, сфера, трапеция, параллелограмм, шестиугольник, пятиугольник, пирамида, конус. Разработайте для них иерархию классов и определите соответствующие наборы функций, использующие наследование для вычисления объема, периметра и площади поверхности этих объектов (где это возможно).

Глава 17
Аппарат исключений

В этой главе сначала описываются основные характеристики и понятия, связанные с организацией аппарата и обработкой исключений, в частности предопределенные и определяемые программистом исключения; обработчики (ловушки) исключений и возникновение исключений. Затем формулируются и обсуждаются этапы работы с определяемыми исключениями, потоки управления при обработке исключений, организация многоуровневой системы исключений. В оставшейся части главы приводятся и анализируются средства обработки исключений в четырех языках программирования: Ada, C++, Java и C#.

Характеристика исключений

В любой жизнеспособной системе должна предусматриваться реакция на чрезвычайные (исключительные) ситуации. Такую реакцию в языке программирования предусматривает аппарат исключений [3, 12, 13, 26, 31, 32, 34, 38, 44, 52, 76].

Требования к аппарату исключений:

1. Полнота исключений — на любое исключение должна быть предусмотрена реакция исполнителя (компьютера).
2. Минимальность возмущений — затраты на учет чрезвычайных ситуаций должны быть минимальны.
3. Минимальность повреждений — ущерб при возникновении исключений должен быть минимален.

С теоретической точки зрения требование полноты реализовать невозможно, но практический подход позволяет выделить перечень исключений, которые могут предугадать авторы языка, и дополнительный перечень, предсказываемый разработчиками некоторого программного обеспечения (ПО). Авторы языка программирования опираются на исследование комплекса проблем, выявленных программистским сообществом при написании любых компьютерных программ. Разработчики ПО исходят из своего опыта по созданию (и эксплуатации) конкретных программных приложений. Таким образом, очерчивают круг возможных неприятностей, на которые целесообразно отреагировать.

Минимальность возмущений можно интерпретировать следующим образом: желательно, чтобы заботы об исключениях в минимальной степени сказывались на всех этапах создания и использования программных продуктов.

Обсуждая минимальность повреждений, говорят о способности аппарата как можно раньше и как можно точнее реагировать на исключение, чтобы предотвратить дальнейшее разрушение программы и данных в аварийных условиях.

Исключение — это необычное, аварийное событие (исключительная ситуация), которое обнаруживается аппаратно или программно и требует специальной обработки (обработки исключения).

Обработка производится *обработчиком исключения* (*ловушкой исключения*).

Различают предопределенные и определяемые программистом исключения.

Предопределенные исключения встроены в язык программирования, они фиксируют ошибки ограничений, переполнения, деления на нуль, исчезновение порядка. Их полнота гарантирует, что при нарушении любого языкового требования создается конкретное предопределенное исключение. Минимальность возмущений проявляется в возникновении предопределенного исключения без указания программиста.

Например, в языке Ada, который справедливо считают самым надежным языком программирования, предопределенные исключения образуют:

- *ошибка ограничения* — `Constraint_Error` — возникает при нарушении допустимого диапазона значений или индексов;
- *программная ошибка* — `Program_Error` — возникает при некорректном поведении программы (в языке выделяют два случая: `Bounded_Error` — выход поведения за допустимые границы; `Erroneous_Execution` — проявление причин, ведущих к неправильному выполнению);
- *недостаток памяти* — `Storage_Error` — возникает при нехватке памяти для размещения динамических объектов;
- *ошибка взаимодействия* — `Tasking_Error` — возникает при нарушениях во взаимодействии асинхронных (параллельных) процессов.

Программная ошибка может возникнуть во множестве ситуаций, где программа некорректна, но компилятор не может выявить это во время компиляции. К таким ситуациям относятся: возможность достижения конца функции без выполнения операции `return`; проверка доступности для ссылочных типов во время выполнения.

Следствия использования предопределенных исключений:

1. Программист не должен разрабатывать возникновение таких исключений.
2. Соответствующие указания не загромождают программу.
3. Соответствующие проверки реализуются аппаратурой или авторами компиляторов.

Пример. Рассмотрим блок (составной оператор) с меткой B:

```
<<B>>  -- это обозначение метки
declare
    A : Float; -- в декларативном разделе объявляются локальные переменные
begin
    A := X * X;
    Y := A * EXP(A);   -- возможно переполнение;
    -- можно перекрыть реакцию на ошибку
exception        -- ловушка исключений;
    when Constraint_Error =>
        Y := Float'Last;  -- наибольшее вещественное
    Put ("Переполнение при вычислении Y в блоке B");
end B;
```

Здесь ловушка исключений — это определенная программистом реакция на предопределенное исключение `Constraint_Error` (она может отсутствовать, так как в системе есть стандартная ловушка).

Определяемые исключения расширяют возможности аппарата исключений. Они ориентированы на ошибки, которые не могли предугадать авторы языка, но о которых знают разработчики конкретных программных систем. Эти исключения явно определяются программистом с помощью объявления.

Например, в языке Ada определяемые исключения декларируются программистом с помощью следующих объявлений:

```
< ИмяИсключения > : exception;
```

Пример. Объявим исключения:

```
Объект_Пуст, Ошибка_В_Данных : exception;
```

Мы создали описания двух исключений. Программист должен задать хотя бы одну реакцию на объявленное исключение (построить ловушку — тело, реализацию обработчика исключения). Само определяемое исключение возникает по оператору `raise` в программе. Этот оператор записывает программист, предваряя его «запуск» анализом возможной ситуации. Например, в результате оператора

```
if <входные_данные_очень_подозрительны> then raise Ошибка_В_Данных end if;
```

возникает исключение `Ошибка_В_Данных`.

При возникновении исключения исполнитель переводится в режим обработки исключения:

1. Происходит распространение исключения (ищется подходящая ловушка);
2. Выполняется «реакция на исключение», описанная в найденной ловушке.

Обычно все обработчики (ловушки) исключений группируются в начале или в конце большой программы или подпрограммы, где может произойти исключительная ситуация. Типичной в этом отношении является структура языка Ada, где обработчики «прижимаются» к закрывающей скобке какого-нибудь блока:

```
procedure Sub is
Bad_Data_Value: exception;
            --другие объявления в Sub
begin
            -- операторы, выполняемые при нормальном ходе Sub
exception
       when Bad_Data_Value =>
             -- обработчик для некорректных значений данных
       when Constraint_Error =>
             -- обработчик для предопределенного исключения Constraint_Error
       when others =>
             -- обработчик для всех других исключений
end Sub;
```

В данном примере обработчик исключений (он начинается с ключевого слова `exception`) располагается в конце раздела операторов процедуры `Sub`. Внутри обработчика размещено несколько вариантов обработки. Каждый вариант начинается с ключевого слова `when`, за которым записывается имя анализируемого исключения.

Обработчик может предусматривать реакцию на несколько исключений. Форма записи такого обработчика имеет вид:

```
...
exception        -- начало ловушки;
    when Program_Error | Был_Невежлив =>
            Put ("Извинись и исправь!");
    when ...
    when others => Put ("Фатальная ошибка!");
end;
```

Видим, что в отдельном варианте выбора имена исключений перечисляются через вертикальную черту. Выбор `others` в перечне вариантов должен быть последним. Он конкретизирует реакцию на все оставшиеся исключения.

Этапы работы с определяемыми исключениями

В жизненном цикле программы с определяемыми исключениями условно можно выделить четыре этапа. Проиллюстрируем эти этапы (рис. 17.1).

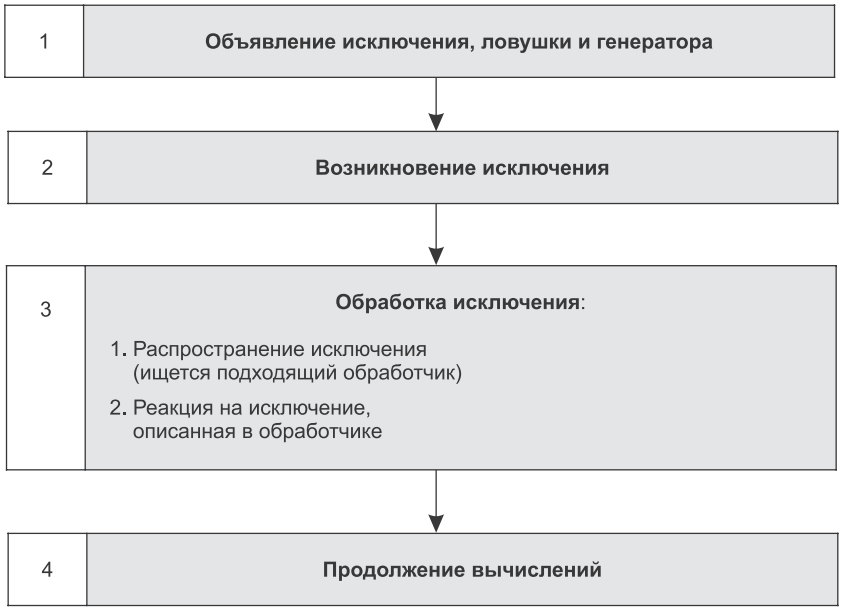


Рис. 17.1. Этапы работы с определяемыми исключениями

На *первом этапе*, при создании программы программист объявляет исключение, пишет под него ловушку и вводит в программу «генератор» исключения, который анализирует корректность хода вычислений и при необходимости может сгенерировать предварительно объявленное исключение.

Второй этап — это этап вычислений программы. Здесь может произойти авария и возникнуть сигнал исключения. В этой ситуации нормальные вычисления приостанавливаются и система переключается в режим обработки исключений, переходя к третьему этапу.

На *третьем этапе* в вычислительной среде обычные действия «заморожены», движется лишь сигнал исключения. Считается, что он движется в направлении, где его поджидает ловушка (обработчик) исключения. Говорят, что исключение распространяется от точки его порождения до точки его обработки. *Обработчик* — это специализированная подпрограмма, которая реагирует на аварию (конкретное исключение), в максимальной степени демпфируя ее последствия.

Правило, определяющее, какой обработчик должен обрабатывать конкретное исключение, формулируется в терминах *динамической цепи активаций* подпрограмм, которая ведет к подпрограмме, породившей исключение. Если исключение `X` порождено в подпрограмме `A`, то оно обрабатывается обработчиком, определенным в этой подпрограмме `A` (если таковой имеется). Если обработчика нет, то `A` завершает свое выполнение. Если `A` была вызвана из подпрограммы `B`, то исключение распространяется на `B`. Если в `B` нет обработчика для исключения `X`, то `B` также завершается, а исключение распространяется на подпрограмму `C`, вызвавшую `B`, и т. д. Если ни в одной подпрограмме, а также в главной программе не найдется обработчика для этого исключения, то вся программа останавливается и вызывается стандартный обработчик, определенный в данном языке программирования

Важно подчеркнуть, что эта логика распространения исключений позволяет подпрограмме выполнять обычную работу даже в аварийной ситуации. Элементарная операция или подпрограмма может внезапно прервать свое нормальное выполнение и породить исключение. Для вызывающей подпрограммы эффект порождения исключения подпрограммой и элементарной операцией будет одинаковым, если подпрограмма сама не обрабатывает это исключение. Если же это исключение обрабатывается внутри подпрограммы, то она завершается нормальным образом, и вызывающая подпрограмма даже не заметит, что имела место исключительная ситуация.

Итак, при распространении исключения используется *принцип динамической ловушки* — выбирается обработчик, динамически ближайший к месту происшествия.

В качестве примера рассмотрим программу:

```
procedure Main is
    Ошибка : exception;  -- описание исключения
    procedure Inner is
    begin
        ... -- ( 1 )
    end Inner;
    procedure Outer is
    begin
        Inner;    -- вызов процедуры;
        ... -- ( 2 )
        exception -- первая ловушка;
        ...
        when Ошибка => Put ("Ошибка в Outer");
        -- реакция на исключение в первой ловушке
    end Outer;
begin     -- Main
    ... -- ( 3 )
    Outer;   -- вызов процедуры;
    exception  -- вторая ловушка;
    when Ошибка => Put ("Ошибка в Main");
    -- другая реакция на то же исключение во 2-й ловушке
end Main;
```

продолжение ⇩

Здесь (1), (2), (3) — возможные места возникновения исключений.

Если исключение `Ошибка` возникнет на месте (3), то сработает реакция во второй ловушке (`"Ошибка в Main"`). Если исключение возникает на месте (2), то есть при выполнении процедуры `Outer`, то сработает реакция в первой ловушке (`"Ошибка в Outer"`). Если исключение возникает на месте (1), при выполнении процедуры `Inner`, в которой ловушки нет, то динамический выбор приведет к первой ловушке (из тела `Outer`). Статический выбор обеспечил бы реакцию из второй ловушки, что менее точно.

Можно сделать вывод: исключения объявляются статически, а реакция на них выбирается динамически.

После обработки исключения переходят к четвертому этапу — продолжению вычислений. Содержание этого этапа целиком и полностью зависит от реализованной в системе методологии.

Потоки управления при обработке исключений

Опишем логику связывания исключений с обработчиками, а также логику продолжения программы после завершения обработки (рис. 17.2).

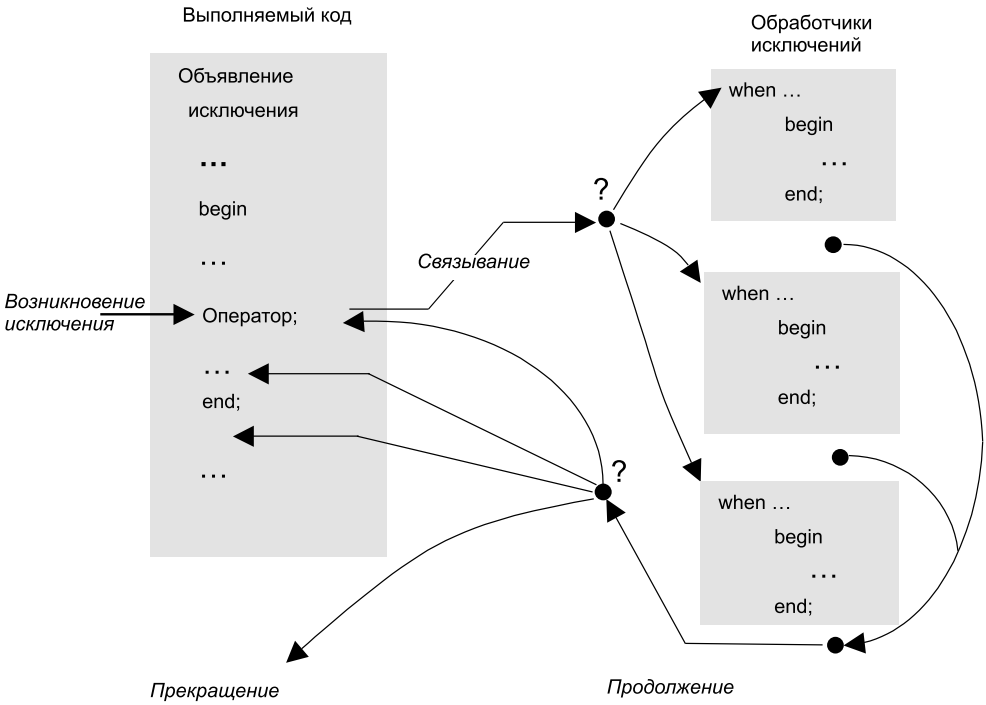


Рис. 17.2. Потоки управления при обработке исключений

Обычно в программе предусматривается целый ряд обработчиков. Каждый обработчик нацелен на свою категорию исключений. Возникший сигнал исклю-

чения связывается с подходящим обработчиком. После завершения своей работы обработчик должен вернуть управление в программу. На рис. 17.2 знаком вопроса отмечены возможные альтернативы:

- возврат к оператору, породившему исключение;
- возврат в блок (подпрограмму), где возникло исключение;
- возврат во внешний, объемлющий блок (подпрограмму), внутри которого находится блок, сгенерировавший исключение;
- прекращение вычислений.

Наиболее популярны два принципа продолжения вычислений:

- принцип пластыря;
- принцип катапульты.

Принцип пластыря. Суть поведения: прервать исполняемый процесс, вызвать врачебную бригаду (она накладывает пластырь на рану), после окончания лечения продолжить прерванный процесс. Недостаток: нет гарантии, что «заклеенный» процесс сможет нормально работать.

Принцип катапульты. В основе принципа катапульты лежит стремление к повышению надежности вычислений. Основная идея — считать появление исключения свидетельством полной непригодности «аварийного процесса» к нормальной работе.

В этом случае используется следующая методика:

1. Последовательно признавать аварийными вложенные процессы (начиная с самого внутреннего).
2. Движение из вложенных в объемлющие процессы продолжать до тех пор, пока не найдется процесс с подготовленной реакцией на исключение.
3. Процессы, в которых нет подготовленной (нужной) реакции на исключение, завершать катапультированием.
4. Найденная реакция должна обеспечить нормальную работу уцелевших процессов.
5. Возврат к аварийному процессу не предусматривается.

Многоуровневая система исключений

Многоуровневая система должна обеспечить реакцию на любые исключения. Например, может быть предусмотрена иерархия, представленная на рис. 17.3.

В подобной системе работа начинается на нижнем уровне, уровне исполнителей. Для исполнителей предусматривается набор всевозможных определяемых исключений:

```
Нет_Исполнителей, Нет_Ресурсов, Нет_Заказов : exception;
-- введены три определяемые исключения
```

Любой исполнитель может отреагировать на ситуацию, генерируя конкретное исключение, но принятие мер откладывается до достижения компетентного уровня иерархии. Исполнитель просто формирует исключение. Например, могут предусматриваться следующие исполнители:

```
procedure Распределить_Работу is
        ...
        raise Нет_Исполнителей;  -- генерируется исключение,
                -- но ловушки нет (неясно, что делать!)
    end Распределить_Работу;

procedure Выполнить is
        ...
        raise Нет_Ресурсов;
            -- генерируется исключение, но ловушки нет
    end Выполнить;

procedure Проверить_Исполнение is
        ...
        raise Нет_Заказов;
            -- генерируется исключение, но ловушки нет
end Проверить_Исполнение;
```

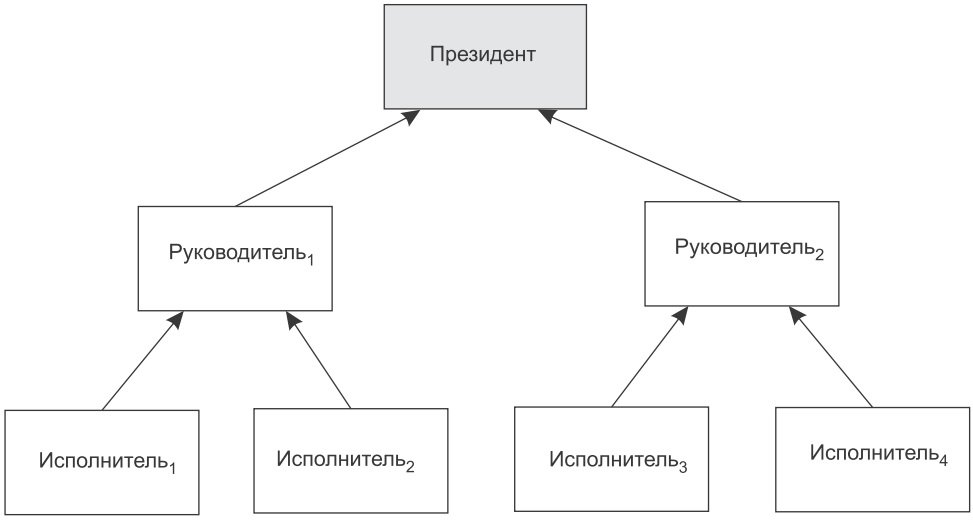


Рис. 17.3. Многоуровневая система исключений

На среднем уровне осуществляется руководство работой исполнителей и отслеживается их реакция на аварийные ситуации, однако информация по возникшим исключениям пересылается наверх, на высший уровень иерархии. Будем считать, что этим занимается руководитель отдела:

```
procedure Руководить_Работой_Отдела ( F : Файл ) is
        -- некомпетентный уровень
    begin
        Открыть ( F );
        Распределить_Работу;
        ...
        Выполнить;
        ...
        Проверить_Исполнение;
        ...
        Закрыть ( F );
```

```
        exception   -- универсальная ловушка для любых исключений
            when others => Закрыть ( F );
                raise -- перенаправление последнего исключения
                    -- к более старшему объемлющему блоку
    end Руководить_Работой_Отдела;
```

Наконец, на высшем уровне президент принимает компетентные решения по всем аварийным ситуациям:

```
procedure Руководить_Работой_Фирмы is
    Нет_Исполнителей, Нет_Ресурсов, Нет_Заказов : exception;
    begin
        ...
        -- будем считать, что это - компетентный уровень
        -- поэтому пишем ловушку для всех реакций
        exception
            when Нет_Исполнителей => Put ("Вас много, а я - один! ");
            when Нет_Заказов => Put("Нет заказов - нужна реклама!");
            when Нет_Ресурсов => Put ("Купи компьютер!");
    end Руководить_Работой_Фирмы;
```

Обработка исключений в языке C++

В этом языке область действия обработчиков исключений задается с помощью блока `try`, а обработка исключений производится с помощью `catch`-функций:

```
try {
//  Программный текст, где возможно возникновение исключений
}
catch  (формальный параметр)  {
//  тело первого обработчика
}
…
catch  (формальный параметр)  {
//  тело  m-го  обработчика
}
```

Блок `try` содержит набор операторов, образующий фрагмент программного текста, при выполнении операторов которого могут возникнуть исключения. Эти исключения распространяются в направлении закрывающей скобки блока. За закрывающей скобкой располагаются `catch`-функции, реагирующие на исключения в этом блоке `try`. Каждая `catch`-функция может иметь только один формальный параметр, который включает имя типа и переменной или только имя типа. Когда информация об исключении передается в обработчик, параметр содержит тип и имя переменной, используемой для этой цели. Если же в формальном параметре `catch`-функции указан только тип, обработчик реагирует на все переменные этого типа. Обработчик с пустым списком формальных параметров «ловит» все исключения.

Связывание исключений с обработчиками

Само исключение в языке C++ возникает по оператору

```
throw  [ <выражение> ]
```

Тип выражения должен совпадать с типом формального параметра какого-то обработчика исключений. Квадратные скобки являются метасимволами, указывающими на необязательность выражения. Оператор `throw` без операнда может появиться только в обработчике. При этом он повторно возбуждает исключение, которое затем обрабатывается где-нибудь в другом месте. Это полностью повторяет использование оператора `raise` в Ada без указания имени исключения.

По типу выражения `throw` выбирается конкретный обработчик, имеющий соответствующий тип формального параметра. Более точно: обработчик с формальным параметром типа `T`, `const T`, `T &` (ссылка на объект типа `T`) или `const T &` соответствует оператору `throw` с выражением, имеющим тип `T`. Если же тип `T` обозначает класс, то «соответствующим» считают обработчик, параметр которого имеет тип `T`, или любой класс, являющийся предком класса `T`.

При возникновении исключения вычисления в блоке `try` прекращаются. Происходит поиск подходящего обработчика. Поиск начинается с обработчика, который непосредственно примыкает к блоку `try`. Если совпадения нет, исключение распространяется на объемлющий блок (из которого вызывалась функция, где появилось исключение). Последний обработчик часто записывается с формальным параметром «многоточие» (. . .), который соответствует любому исключению. Это гарантирует, что все исключения будут обработаны.

После нормального завершения работы обработчика управление передается первому оператору следующего блока `try`.

Если в программе не найден подходящий обработчик, тогда вызывается стандартный обработчик `unexpected`, который просто останавливает программу. Он может быть замещен пользовательским обработчиком, функцией с заголовком `void set_terminate( )`.

В языке C++ нет предопределенных исключений, но стандартные библиотеки определяют и генерируют исключения, такие как `out_of_range`, которое может быть создано контейнерными библиотечными классами, и `overflow_error`, которое может быть сформировано библиотечными математическими функциями.

Оформление функций в C++

Функция в языке C++ может перечислять типы возможных исключений. Для этого в заголовок функции добавляется зарезервированное слово `throw` с последующим списком этих типов, заключенным в круглые скобки. Например, заголовок функции

```
int func( ) throw (int, char *)
```

указывает, что функция `func( )` может возбудить исключения типа `int` или `char *`, и никакие другие. Цель этого добавления — предупредить пользователей функции о возможных исключениях, генерируемых функцией. Предложение `throw` считается публикацией контракта между функцией и ее пользователями. Контракт гарантирует, что другие исключения при выполнении функции не возникнут. Если функция создаст какое-либо другое исключение, программа будет прекращена. Компилятор игнорирует предложения `throw`.

Если типы в предложении `throw` являются классами, то функция может создать любое исключение, производное от перечисленных классов. Если функция возбуждает исключение, не указанное в списке исключений и не являющееся произ-

водным от указанного там, то вызывается стандартный обработчик. Список типов в `throw` может быть пустым. Это означает, что функция не генерирует ни одного исключения. Если в заголовке функции нет предложения `throw`, то функция может инициировать любое исключение. Список типов исключений не считается частью объявления функции.

Если некоторая функция переопределяет функцию с предложением `throw`, то переопределенная функция не может увеличивать список исключений.

Обработка исключений в языке Java

Обработка исключений в Java основана на механизме исключений C++, но в большей степени соответствует объектно-ориентированному подходу. Кроме того, Java включает набор предопределенных исключений, которые неявно генерируются виртуальной машиной Java (Java Virtual Machine — JVM).

Классы исключений

Все исключения в языке Java являются объектами классов, производных от класса `Throwable`. Система языка Java содержит два предопределенных класса исключений, являющихся подклассами класса `Throwable`: `Error` и `Exception`. Класс `Error` и производные от него классы связаны с исключениями категории «исчерпана динамическая память», порождаемыми интерпретатором языка Java. Эти исключения никогда не возбуждаются пользовательскими программами и не обрабатываются ими. Предусмотрены два прямых потомка класса `Exception` — это классы `RuntimeException` и `IOException`. Как следует из названия класса, исключение `IOException` возникает при ошибках в операциях ввода или вывода. Эти операции определены в виде методов в различных классах, объявленных в пакете `java.io`.

Существуют предопределенные дочерние классы, являющиеся потомками от класса `RuntimeException`. В большинстве случаев исключение класса `RuntimeException` возникает, когда ошибку порождает пользовательская программа. Например, обычно генерируется исключение-экземпляр класса `ArrayindexOutBoundsException`, определенного в пакете `java.util` и являющегося наследником класса `RuntimeException`. Часто создается исключение-объект класса `NullPointerException`, другого потомка класса `RuntimeException`.

Пользовательская программа может определять свои собственные классы исключений. В языке Java принято соглашение о том, что определенные пользователем исключения задаются подклассами класса `Exception`.

Обработчики исключений

Обработчики исключений в языке Java имеют тот же вид, что и в языке C++. Однако каждая `catch`-функция должна иметь параметр, а класс параметра должен быть потомком от предопределенного класса `Throwable`.

Синтаксис и смысл конструкции `try` в языке Java почти такой же, что и в языке C++, за исключением секции `finally`, которая может замыкать список обработчиков за конструкцией `try`.

Связывание исключений с обработчиками

Генерация исключения происходит довольно просто. Экземпляр класса исключения является операндом оператора `throw`. Определим класс исключения с именем `TheException` следующим образом:

```
class TheException extends Exception {
    public TheException() {}
    public TheException(String message) {
        super (message);
    }
}
```

Теперь можно сгенерировать исключение:

```
throw new TheException();
```

Создать экземпляр исключения можно и отдельно от оператора `throw`, как показано ниже:

```
TheException theExceptionObject = new TheException();
. . .
throw theExceptionObject;
```

Один из двух конструкторов, включенных в наш новый класс, не имеет параметров, а другой имеет параметр в виде объекта типа `String`, который он передает в суперкласс (`Exception`) для отображения. Следовательно, наше новое исключение может быть сгенерировано следующим образом:

```
throw new theException("сообщение, указывающее место ошибки");
```

Связывание исключений с обработчиками в языке Java подобно языку C++. Если исключение появилось внутри конструкции `try`, оно перемещается за закрывающую скобку конструкции и связывается с первым обработчиком (`catch`-функцией), класс формального параметра у которого такой же, что у объекта-исключения. Разрешается, чтобы класс параметра обработчика был предком класса исключения. Далее связанный обработчик выполняет обработку исключения.

Исключения могут быть обработаны, а затем снова сгенерированы. Для этого достаточно включить оператор `throw` без операнда в конец обработчика. Повторно выработанное исключение уже не будет обрабатываться на уровне первоначальной конструкции `try`, так что зацикливание не возникает. Повторная генерация предусматривается, когда следует выполнить некоторое локальное действие, а затем дальнейшую обработку перенести на уровень объемлющей конструкции `try` или вызывающего модуля. Оператор `throw` в обработчике может также создать исключение другого класса. Следовательно, одно исключение способно породить другое исключение.

Для проверки того, что исключения, созданные в конструкции `try`, всегда обрабатываются некоторым методом, можно ввести специальный обработчик, подходящий для исключений всех классов, производных от класса `Exception`. Достаточно определить обработчик с параметром типа `Exception`:

```
catch (Exception anyObject) {
. . .
}
```

Поскольку имя класса всегда соответствует самому себе или любому классу-предку, любой класс, являющийся потомком класса `Exception`, соответствует

классу `Exception`. Конечно, такой обработчик должен всегда помещаться в конец списка, замыкающего конструкцию `try`, так как он блокирует использование любого последующего обработчика. Следовательно, поиск подходящих обработчиков выполняется последовательно и заканчивается, когда соответствие найдено.

Секция finally

Часто возникает ситуация, когда необходимо выполнить фрагмент программы вне зависимости от того, было или нет сгенерировано исключение внутри конструкции `try`. Обычно это имеет отношение к операциям, не связанным с освобождением памяти (так как работа с динамической памятью входит в обязанности сборщика мусора). Скажем, нужно вернуть в исходное состояние что-то другое, а не память. Речь может идти, например, об открытом файле или сетевом подключении, части изображения на экране или даже каком-то физическом переключателе.

Для достижения желаемой цели после всех обработчиков исключений, примыкающих к конструкции `try`, размещается секция `finally`.

В общем случае полная конструкция `try` и секция `finally` выглядят следующим образом:

```
try {
. . .
}
catch (. . .) {
. . .
}
. . . //** другие обработчики
finally {
. . .
}
```

Семантика этой конструкции следующая:

1. Если в конструкции `try` не возникло ни одного исключения, секция `finally` выполняется перед тем, как выполнение программы будет продолжено в следующей конструкции `try`.
2. Если в `try` появилось некоторое исключение и оно перехвачено обработчиком, секция `finally` выполняется после завершения работы обработчика.
3. Если в `try` сгенерировано исключение, но оно не перехвачено последующим обработчиком, секция `finally` выполняется до распространения исключения на новый уровень.

Секция `finally` может следовать и за конструкцией `try`, после которой обработчиков исключений просто нет. Это имеет смысл лишь в ситуациях, когда внутри блок `try` содержит операторы `throw`, `break`, `continue` или `return`. Цель секции в данных ситуациях та же, что и при обработке исключений. Например:

```
try {
    for (i = 0; i < 250; i++) {
        . . .
        if (. . . ) {
            return;
        } //** конец оператора if
        . . .
```

продолжение ⇩

```
    } //** конец цикла for
} //** конец конструкции try
finally {
    . . .
} //** конец секции finally
```

Секция `finally` здесь будет выполнена независимо от того, прекратил ли оператор `return` выполнение цикла, или он завершился нормальным образом.

Обработка исключений в языке C#

Аппарат исключений в C# предоставляет структурированный, унифицированный и безопасный с точки зрений типов способ обработки ошибок как на системном уровне, так и на уровне пользовательских программ. Механизм исключений в C# достаточно сильно похож на используемый в C++, однако существует несколько важных отличий:

1. В C# все исключения должны быть представлены объектами — экземплярами класса, унаследованного от класса `System.Exception`. В C++ для представления исключения можно использовать значение любого типа.
2. В C# можно использовать секцию `finally` для написания завершающего кода, который выполняется как при нормальной работе программы, так и при возникновении исключений. В C++ это сделать довольно сложно без дублирования кода.
3. В C# исключения системного уровня, такие как переполнение, деление на ноль и разыменование `null`, имеют четко определенные классы исключений и находятся наравне с состояниями ошибок уровня пользовательской программы.

Как видим, аппарат исключений C# очень близок к аналогичному аппарату в языке Java.

Контрольные вопросы и упражнения

1. Выполните детальное сравнение средств для обработки исключений в языках Ada и C++.
2. Напишите программу на языке C++, которая вычисляет сумму квадратов списка целых чисел. Числа вводятся с клавиатуры, их значения должны находиться в диапазоне от –50 до +50. Предусмотрите обработку исключений, чтобы гарантировать, что вводимые числа принадлежат указанному диапазону и являются целыми. Дополнительно обработайте исключение, генерируемое, если результат выходит за пределы значений для типа `integer`.
3. Напишите программу для устойчивого к ошибкам ввода целых и вещественных чисел. Процедуры программы должны обеспечивать проверку величины и типа набираемого значения. Величина числа должна находиться внутри определенного диапазона. Границы диапазона должны задаваться при вызове процедуры. Ввод должен повторяться до тех пор, пока не будет набрано правильное число. При неправильном вводе должны генерироваться исключения.

Глава 18
Ввод-вывод и файлы

Эта глава посвящена файлам, потокам и организации ввода-вывода. После пояснения задач аппарата ввода-вывода и классификации файлов здесь обсуждаются средства как процедурного, так и объектно-ориентированного (потокового) ввода-вывода.

Ввод-вывод — это передача данных между компьютером и внешней средой. Источник всех проблем при вводе-выводе — сложность и неопределенность внешней среды (внешнего мира) [12, 13, 17, 20, 26, 34, 62].

Перечислим особенности внешней среды с точки зрения ввода-вывода.

Относительная независимость внешних объектов от исполнителя. Внешние объекты (и связи между ними) не находятся под полным контролем компьютера. Поэтому в языке программирования должны быть средства для описания взаимодействия с объектами заранее неизвестной природы. Примерами таких средств могут быть:

- аппарат определения новых типов;
- аппарат управления конкретным представлением объектов;
- способ представления внешних имен (например, строкой символов, заключенной в кавычки).

Разнообразие внешних устройств. Состав партнеров компьютера по вводу-выводу зависит от предметной области (табл. 18.1) и от решаемой задачи (табл. 18.2).

Таблица 18.1. Зависимость от предметной области

Область	Ввод	Вывод
Числовые расчеты	С магнитных дисков	На печать
Игры	С клавиатуры, мыши	На цветной экран
Управление в реальном времени	От датчиков	На исполнительные органы управления объектами

Таблица 18.2. Зависимость от решаемой задачи

Задача	Ввод	Вывод
Начальный набор текста	С клавиатуры	На экран
Редактирование	С мыши	На принтер
Оформление макета книги	С диска	На наборную машину

Кроме того, состав внешних устройств может изменяться в ходе решения задачи (датчики, исполнительные устройства, приборы могут выходить из строя, может меняться конфигурация аппаратуры).

Для учета изменчивости внешней среды в языке программирования нужен соответствующий аппарат абстракции-конкретизации. Абстракции обеспечивают моделирование внешней среды в программе. При этом игнорируется специфика потенциальных внешних устройств. Конкретизация обеспечивает настройку на конкретное устройство, которая выполняется в ходе связывания с этим устройством.

Человеческий фактор. Партнером по вводу-выводу может быть человек со своими свойствами и требованиями. Пример свойства человека: способность ошибаться при вводе данных. Пример требования: необходимость представить данные в удобном для чтения виде. Отсюда необходимость в средствах форматирования, управления графикой, цветом, звуком (для воздействия на человека). Обычно в языке программирования есть простейшие средства форматирования, все остальное должно программироваться с помощью средств развития.

Динамизм и относительная ненадежность. Полный статический контроль внешнего устройства невозможен (в отличие от внутренних объектов программы). Например, у внутреннего объекта — целой переменной — не может быть вещественного значения, так как нет способа выполнить в программе такое присваивание. Однако язык программирования не может запретить человеку ошибаться или испортить внешний носитель файла. Поэтому при вводе-выводе нужен динамический контроль с помощью исключений.

Параллелизм. Жизненный ритм внешних устройств может быть независим по отношению к компьютеру. Поэтому в языке программирования должен быть аппарат управления асинхронными процессами (аппарат прерываний, многозадачности и т. д.).

Характеристика аппарата ввода-вывода

Задачи аппарата ввода-вывода языка программирования:

- именование внешних объектов;
- связывание внешних объектов с внутренними объектами;
- контроль и управление исключениями;
- форматирование;
- подключение устройств заранее неизвестной природы.

Аппарат ввода-вывода должен оберегать программиста от особенностей конкретного устройства ввода-вывода. Обычно язык программирования содержит *абстрактную машинно-независимую модель ввода-вывода*. Ею пользуется программист. Все проблемы связывания (учета реальной внешней среды) решаются вне программы. Для этого используются средства операционной системы (связывающие программы-драйверы).

Основное понятие абстрактной модели ввода-вывода — это *логическое внешнее устройство* (ЛВУ). Оно является моделью определенной категории физических

устройств, отражает существенные особенности категории. Конкретно — возможность применения к устройству операций ввода-вывода данных определенного типа.

С другой стороны, в ЛВУ не отражены физические характеристики категории устройств (нет дорожек и цилиндров магнитного диска, зон магнитной ленты).

Наиболее распространенный вариант модели ввода-вывода — *файловая модель*. В ней все внешние устройства представлены файлами. Файл — это именованный объект предопределенного типа. К файлам применим определенный набор операций.

Файл — это структура данных, обладающая двумя свойствами:

- Она располагается на каком-либо внешнем запоминающем устройстве (магнитная лента, диск), и ее размер может быть значительно больше, чем размеры других типов структур данных.
- Время жизни файла может значительно превышать время выполнения создавшей его программы.

Наиболее распространенной разновидностью файлов считают *файлы последовательного доступа*, но используются также и *файлы прямого доступа*. Файлы последовательного доступа позволяют получать данные только в том порядке, в котором они ранее были переданы. В файлах прямого доступа есть возможность изменять этот порядок, управляя позицией, в которой выполняется ввод-вывод. Эта позиция определяется индексом (аналогом адреса внутренней памяти).

Прежде чем мы перейдем к обсуждению файлов, рассмотрим стандартные средства ввода-вывода на примере языка Ада.

Пакеты ввода-вывода языка Ада

Ада — это язык для разработки программ реального мира, программ, которые могут быть очень большими (включают сотни тысяч операторов). При этом желательно, чтобы отдельный программный модуль имел разумно малый размер. Для обеспечения такого ограничения Ада построена на идее библиотек и пакетов. В библиотеку, составленную из пакетов, помещаются программные тексты, предназначенные для многократного использования.

Пакет подключается к программе с помощью указания (спецификатора) контекста, имеющего вид: `with < Имя_Пакета >;`

В частности, в пакетах размещены процедуры ввода-вывода для величин предопределенных типов (табл. 18.3).

Таблица 18.3. Пакеты ввода-вывода

Имя пакета	Тип вводимых-выводимых величин
Ada.Text_IO	Character, String
Ada.Integer_Text_IO	Integer
Ada.Float_Text_IO	Float

Для поддержки ввода-вывода величин других типов, определяемых пользователем, используются шаблоны (заготовки) пакетов — родовые пакеты. Родовой пакет перед использованием должен быть настроен на конкретный тип.

Родовые пакеты ввода-вывода всегда находятся внутри пакета `Ada.Text_IO` (табл. 18.4).

Таблица 18.4. Родовые пакеты ввода-вывода

Внутренние пакеты библиотеки Ada.Text_IO	
Имя родового пакета	**Нужна настройка на тип**
Integer_IO	Любой целый тип со знаком
Float_IO	Любой вещественный тип с плавающей точкой
Enumeration_IO	Любой перечисляемый тип
Fixed_IO	Любой двоичный вещественный тип с фиксированной точкой
Decimal_IO	Любой десятичный вещественный тип с фиксированной точкой
Modular_IO	Любой целый тип без знака

Для обращения к внутренним родовым пакетам используют составные имена. Например, `Ada.Text_IO.Modular_IO`, `Ada.Text_IO.Enumeration_IO`.

Процесс настройки родового пакета называют конкретизацией. В результате конкретизации образуется экземпляр родового пакета. Экземпляр помещается в библиотеку и может быть подключен к любой программе.

Конкретизация задается предложением следующего формата:

```
with <Полное имя родового пакета>;
        package <Имя пакета-экземпляра> is
            new <Имя родового пакета> ( <Имя типа настройки> );
```

Например, введем перечисляемый тип `Type Summer is ( may, jun, jul, aug );`

Создадим экземпляр пакета для ввода-вывода величин этого типа:

```
with Ada.Text_IO.Enumeration_IO;
        package Summer_IO is new Ada.Text_IO.Enumeration_IO ( Summer );
```

Теперь пакет `Summer_IO` может использоваться в любой программе.

Процедуры ввода языка Ада

Формат вызова процедуры:

```
< ИмяПакета > . < ИмяПроцедуры > ( < ФактАрг > );
```

Перечислим (и охарактеризуем) возможные процедуры ввода.

Процедура ввода величины типа `Character`.

Оператор вызова: `Ada.Text_IO.Get ( Var );`

Описание: Переменной `Var` (типа `Character`) присваивается значение символа, введенного с клавиатуры. Пробел считается символом, нажатие на клавишу `Enter` не учитывается.

Процедура ввода величины типа `String`.

Оператор вызова: `Ada.Text_IO.Get ( Var );`

Описание: Переменная `Var` должна иметь тип `String (min..max)`, где `1<=min<=max`. С клавиатуры читается точно `max-min+1` символов. Нажатие на

клавишу Enter не учитывается как ввод символа. Компьютер будет ждать ввода заданного количества символов (без учета нажатий на Enter).

Процедура ввода величины типа String.

Оператор вызова: Ada.Text_IO.Get_Line (Var, Number);

Описание: Переменная Var должна иметь тип String (min..max). Get_Line пытается прочитать max-min+1 символов. Чтение прекращается при нажатии Enter. После операции Get_Line переменная Number содержит реальное количество прочитанных символов. Если строковая переменная только частично заполнена операцией, то оставшиеся символы не определены.

Процедура ввода величины типа Integer.

Оператор вызова: Ada.Integer_Text_IO.Get (Var);

Описание: В переменную Var заносится строка числовых символов. Все ведущие пробелы и нажатия на Enter игнорируются. Первым отличным от пробела символом может быть знак (+, -) или цифра. Строка данных прекращается при вводе нечислового символа или нажатии Enter.

Процедура ввода величины типа Float.

Оператор вызова: Ada.Float_Text_IO.Get (Var);

Описание: В переменную Var заносится строка символов. Все ведущие пробелы и нажатия на Enter игнорируются. Первым отличным от пробела символом может быть знак (+, -) или цифра. Строка символов может содержать десятичную точку, окруженную числовыми символами. Строка данных прекращается при вводе символа, который не может быть частью числа с плавающей точкой, или нажатии Enter.

Процедура ввода величины перечисляемого типа.

Оператор вызова: Instance.Get (Var);

Описание: Под Instance подразумевается экземпляр родового пакета Enumeration_IO, настроенный на конкретный перечисляемый тип. В переменную Var заносится строка символов. Все ведущие пробелы и нажатия на Enter игнорируются. Первый вводимый символ должен быть буквой, а символы должны формировать один из идентификаторов типа. Ввод прекращается при наборе символа, не принадлежащего идентификатору, или нажатии Enter. Если прочитанный символ не является ни одним из символов перечисляемого типа, то возбуждается Data_Error.

Процедура перехода на новую строку ввода.

Оператор вызова: Ada.Text_IO.Skip_Line;

Описание: Игнорируются оставшиеся символы текущей строки ввода (очищается буфер ввода). Выполняется принудительный переход к началу новой строки ввода.

Процедуры вывода языка Ада

Язык Ада позволяет указывать в операторе вызова не только фактические параметры, но и сопоставления формальных и фактических параметров в следующем формате:

```
< ИмяПроцедуры > ( < ФактПарам1 >,
        < ФормПарам3 > => < ФактПарам3 >, ... );
```

При такой форме порядок записи параметров безразличен. Перечислим (и охарактеризуем) возможные процедуры вывода.

Процедура вывода величины типа `Character`.

Оператор вызова: `Ada.Text_IO.Put ( Item => Var );`

Описание: Значение переменной `Var` (типа `Character`) отображается на дисплее, а курсор перемещается в следующую позицию.

Процедура вывода величины типа `String`.

Оператор вызова: `Ada.Text_IO.Put ( Item => Var );`

Описание: Переменная `Var` должна иметь тип `String (min..max)`, где `1<=min<=max`. На экране отображается точно `max-min+1` символов, а курсор перемещается в очередную позицию после конца строки.

Процедура вывода величины типа `Integer`.

Оператор вызова: `Ada.Integer_Text_IO.Put ( Var, Width => 4 )`

Описание: Отображается значение переменной `Var`, используются текущие `Width` позиций (в примере — 4) на экране. Если значение (включая знак) занимает меньше, чем `Width` позиций, ему предшествует соответствующее количество пробелов. Если значение занимает больше, чем `Width` позиций, то используется реальное количество позиций. Если параметр `Width` пропущен, то используется ширина, заданная компилятором по умолчанию.

Процедура вывода величины типа `Float`.

Оператор вызова: `Ada.Float_Text_IO.Put (Var, Fore=>4, Aft=>2, Exp=>3);`

Описание: Значение переменной `Var` отображается на экране. Параметр `Fore` задает требуемое количество позиций для целой части числа (слева от десятичной точки); параметр `Aft` задает количество позиций для дробной части числа (справа от десятичной точки); параметр `Exp` задает количество позиций в степени (после `E`). Если реальная целая часть числа, включая знак, занимает меньше чем `Fore` позиций, слева добавляются пробелы. Если `Exp=0`, то степень не отображается.

Процедура вывода величины перечисляемого типа.

Оператор вызова: `Instance.Put ( Var, Width => 7 );`

Описание: Отображается значение переменной `Var` (перечисляемого типа), используются `Width` позиций на экране. Если значение будет занимать меньше, чем `Width` позиций, то за ним следует соответствующее количество пробелов. Если значение будет занимать больше, чем `Width` позиций, то используется реальное количество позиций. Если параметр `Width` пропущен, то используется ширина, заданная компилятором по умолчанию.

Процедура перехода на новую строку экрана.

Оператор вызова: `Ada.Text_IO.New_Line (Spacing => <число> );`

Описание: Если параметр `Spacing = 1`, то курсор перемещается в первую позицию следующей строки экрана. Если `Spacing > 1`, то это действие повторяется `Spacing` раз. Если параметр `Spacing` пропущен, то по умолчанию используется 1.

Процедура вывода величины типа `String` с переходом на новую строку экрана.

Оператор вызова: `Ada.Text_IO.Put_Line ( Var );`

Описание: Значение переменной `Var` отображается на экране, после чего курсор перемещается в начало следующей строки экрана.

Организация файлов и средства управления ими

Файл — это структура данных во внешней памяти, имеющая имя и специальную организацию. Во внешней памяти файлы только хранятся. Для обработки данные из файла вызываются в оперативную память. В каждый момент времени возможен доступ только к одному элементу файла. Этот элемент адресуется с помощью указателя файла. Данные из элемента файла заносятся в буферную переменную (рис. 18.1).

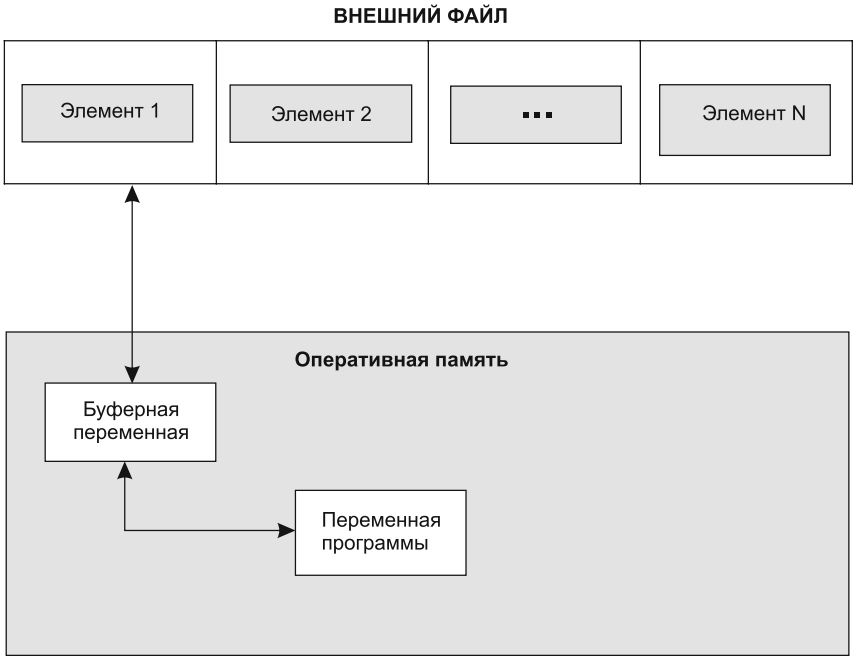


Рис. 18.1. Доступ к элементам файла

После чтения или записи указатель файла перемещается к следующему элементу и делает его доступным для обработки. Количество элементов в файле не фиксируется. Иными словами, файл имеет переменную длину.

Предусмотрены два способа доступа к элементам файла: последовательный и прямой (произвольный).

При последовательном доступе элементы файла доступны только в последовательном порядке (от первого к последнему). Таким образом, перед обращением к `n+1`-му элементу должно произойти обращение к предыдущим `n` элементам файла.

При прямом доступе возможно прямое обращение к любому элементу по его номеру в файле.

Различают понятия: внешний файл и внутренний файл.

Внешний (физический) *файл* — это реальный набор данных, существующий во внешней памяти. *Внутренний файл* — это абстракция, описываемая в программе и обеспечивающая взаимодействие с реальным внешним файлом (рис. 18.2).

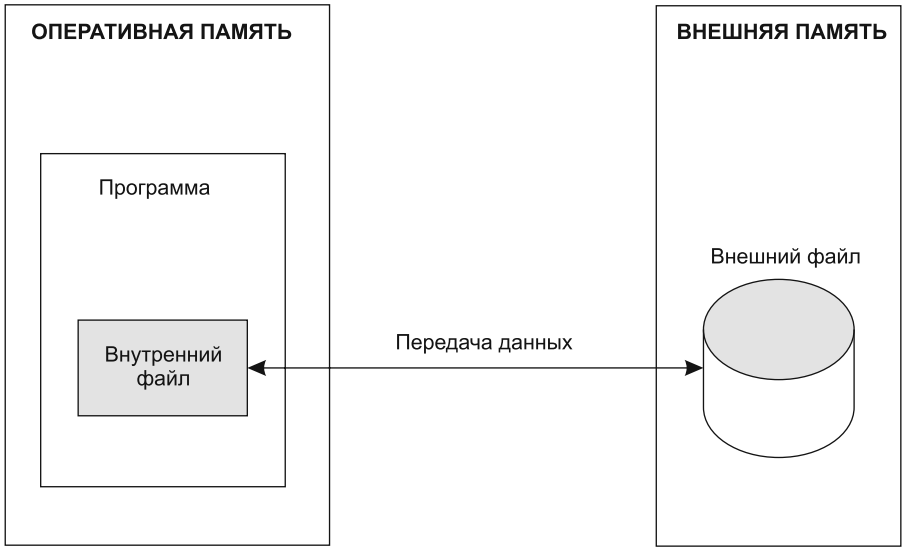


Рис. 18.2. Внешний и внутренний файлы

Программа пользователя не имеет непосредственного доступа к внешнему файлу. Вся работа с внешними файлами описывается в ней в терминах внутренних файлов.

В языке Ада внутренний файл задается как объект файлового типа. Объявление файлового типа находится в стандартных пакетах ввода-вывода и имеет вид:

```
type File_Type is limited private;
```

Из этого объявления следует, что внутреннее содержание файловых объектов закрыто для пользователя. К файловым объектам применимы только те операции, которые определены в пакетах ввода-вывода.

Для управления файлами используются пять операций. Процедура `Create` служит для создания и открытия нового внешнего файла, связывания его с внутренним файлом. Процедура `Open` открывает уже существующий внешний файл и связывает его с внутренним файлом. Процедура `Close` закрывает внешний файл, разрывая его связь с внутренним файлом. Процедура `Delete` удаляет внешний файл. Процедура `Reset` переводит указатель файла в начальную позицию, позволяя возобновить ввод или вывод элементов с самого начала.

Процедура `Create` имеет следующую спецификацию:

```
procedure Create ( File : in out File_Type;
                   Mode : in File_Mode := режим_по_умолчанию;
                   Name : in String := " ";
                   Form : in String := " " );
```

где `File` — имя внутреннего файла, связываемого с создаваемым внешним файлом, `Mode` — режим работы (допустимы значения: `In_File` — ввод, `Out_File` — вывод, `InOut_File` — ввод-вывод, `Append_File` — добавление), `Name` — имя внешнего файла, `Form` — дополнительные характеристики внешнего файла.

Спецификация процедуры `Open` записывается в виде:

```
procedure Open ( File : in out File_Type;
                 Mode : in File_Mode;
                 Name : in String;
                 Form : in String := " " );
```

Формальные параметры у процедуры `Open` те же, что и у процедуры `Create`. Однако параметры `Mode` и `Name` не имеют значений по умолчанию. Дело в том, что запрещается открывать уже открытый файл (возникает исключение `Status_Error`). Если не существует внешнего файла с указанным через `Name` именем, то генерируется исключение `Name_Error` (для процедуры `Create` оно генерируется, если имя внешнего файла недопустимо). Наконец, исключение `Use_Error` вырабатывается, если операционная среда не поддерживает открытие (или создание) внешнего файла по каким-либо другим причинам (например, на диске нет свободного места).

Фактически процедуры создания и открытия готовят внешний файл к сеансу ввода или вывода.

Процедура

```
procedure Close ( File : in out File_Type );
```

завершает сеанс работы с внешним файлом, ее параметром является внутренний файл, связь которого с внешним файлом уничтожается. Если внешний файл не был открыт, то возбуждается исключение `Status_Error`.

Процедура

```
procedure Delete ( File : in out File_Type );
```

уничтожает внешний файл, связанный с данным внутренним файлом.

Процедура `Reset` имеет две разновидности:

```
procedure Reset ( File : in out File_Type;
                  Mode : in File_Mode );
procedure Reset ( File : in out File_Type );
```

С ее помощью внешний файл устанавливается в состояние, позволяющее возобновить ввод-вывод с начала файла или добавление в конец файла (для режима `Append_File`). Если задан параметр `Mode`, то в соответствии с ним устанавливается новый режим работы файла. Так, если надо создать новый файл, записать в него данные, а затем прочитать их и обработать, то необходимы следующие этапы работы:

1. Выполнение процедуры `Create`.
2. Запись данных в открытый файл.
3. Выполнение процедуры `Reset` с заданием режима `In_File`.
4. Чтение и обработка данных из файла.
5. Закрытие файла с помощью процедуры `Close`.

Для получения информации о внешнем файле определены следующие функции:

```
function Mode ( File : in File_Type ) return File_Mode;
-- возвращает характеристику текущего режима работы
function Name ( File : in File_Type ) return String;
-- возвращает имя файла
function Form ( File : in File_Type ) return String;
-- возвращает дополнит. характеристики файла
function Is_Open ( File : in File_Type ) return Boolean;
-- позволяет узнать, открыт ли файл
```

В заключение отметим, что описанные здесь процедуры и функции определены для всех разновидностей файлов Ады.

Текстовые файлы

Текстовый файл является файлом, состоящим из символов. Текстовые файлы считаются основной формой файлов, применяемых пользователем для ввода и вывода данных, так как они могут быть непосредственно распечатаны или введены с клавиатуры. Текстовые файлы — это всего лишь разновидность обычных файлов с последовательным доступом, и с ними можно производить те же действия, что и с «последовательными» файлами.

Средства для работы с текстовыми файлами содержатся в пакетах `Ada.Text_IO`, `Ada.Float_Text_IO`, `Ada.Integer_Text_IO`.

Данные в текстовом файле запоминаются в виде символов. Текстовый файл — это набор символов, группируемых в строки переменной длины. Поэтому способ доступа к элементам текстового файла — только последовательный. Текстовый файл не имеет фиксированного размера. Конец файла отмечается специальным символом `End_Of_File` (обозначается как <eof>). Точная форма представления символа-маркера <eof> зависит от операционной системы.

При создании текстового файла с помощью программы-редактора для разделения файла на строки нажимается клавиша `Enter`. Каждый раз при нажатии на `Enter` в файл помещается другой специальный символ-маркер `End_Of_Line` (обозначается <eoln>).

Обсудим содержание текстового файла, который состоит из двух строк букв, символов пробела и символов пунктуации:

```
Это текстовый файл! <eoln>
Он содержит две строки.<eoln><eof>
```

Каждая строка заканчивается маркером <eoln>. В последней строке за маркером <eoln> следует маркер <eof>. При просмотре содержимого файла каждая строка файла представляется как отдельная строка экрана. В реальном дисковом файле символы запоминаются в последовательности байтов памяти, каждый символ — в своем байте памяти. Байт первого символа второй строки (буква `O`) следует за байтом первого маркера <eoln>.

Текстовый файл может также содержать числовые данные или комбинацию числовых и буквенных данных. Следующий файл включает числовые данные и символы пробела:

```
12345 6789<eoln>
777 -11<eoln><eof>
```

Каждое число записывается на диске как последовательность цифровых символов. Символы пробелов разделяют числа, входящие в одну строку.

В диалоговой программе Ада рассматривает данные, вводимые с клавиатуры, так, как будто они читаются из предопределенного файла `Standard_Input`. Нажатие клавиши `Enter` вводит маркер <eoln> в этот файл. В диалоговом режиме для индикации окончания данных мы обычно используем сигнальную метку, а не пытаемся ввести маркер <eof> в системный файл `Standard_Input`. Впрочем, мы

можем использовать маркер <eof>. Его «клавиатурное» представление зависит от операционной системы. Наиболее часто используются комбинации Ctrl-Z или Ctrl-D.

В свою очередь, отображение символов на экране эквивалентно записи символов в предопределенный файл Standard_Output. Маркер <eoln> помещает в этот файл процедура New_Line. В результате курсор перемещается в начало новой строки экрана. Стандартные файлы Standard_Input и Standard_Output являются текстовыми файлами, содержащими символы.

При работе со стандартными файлами процедуры Create (для Standard_Output) и Open (для Standard_Input) в программе не указываются. Эти процедуры вызываются и выполняются автоматически, по умолчанию. Не ссылаются на эти файлы (с помощью файловых объектов) и в стандартных процедурах ввода-вывода типа Get, Put, Skip_Line, New_Line.

Маркеры <eoln> и <eof> отличны от других символов в текстовом файле, так как не являются символами данных. Фактически в стандарте для Ады они даже не определены, так как их представление зависит от операционной системы. Однако если программа попытается прочитать <eof>, генерируется исключение End_Error.

Для обработки этих маркеров в пакет Ada.Text_IO введены две функции. Эти функции позволяют определить — не является ли следующий символ маркером <eoln> или <eof>.

Функция

```
End_Of_Line ( <ИмяВнутреннегоФайла> )
```

возвращает значение *True*, если следующий символ является маркером <eoln>.

Функция

```
End_Of_File ( <ИмяВнутреннегоФайла> )
```

возвращает значение *True,* если следующий символ является маркером <eof>.

ПРИМЕЧАНИЕ

Для стандартных файлов ввода-вывода аргумент в этих функциях не указывается.

Алгоритм обработки файла данных на основе этих функций имеет вид:

```
while Not End_Of_File ( Имя ) loop
    while Not End_Of_Line ( Имя ) loop
            обработка каждого символа строки
    end loop;
        -- предполагается: следующий символ - <eoln>
        обработка символа <eoln>
end loop;
        -- предполагается: следующий символ - <eof>
```

Если файл данных не пуст, то начальный вызов End_Of_File вернет значение *False* и будет выполняться внутренний цикл. Этот цикл обрабатывает каждый символ в строке, за исключением <eoln>. Например, для представленного выше 2-строчного файла первое выполнение внутреннего цикла обработает первую строку:

```
Это текстовый файл!
```

Как только следующим символом станет `<eoln>`, функция `End_Of_Line` вернет значение *True* и внутренний цикл прекратится. Маркер `<eoln>` обрабатывается во внешнем цикле сразу после выхода из внутреннего цикла.

Каждое повторение внешнего цикла начинается с вызова функции `End_Of_File` для проверки — не является ли следующим символом маркер `<eof>`. Если это правда, функция возвращает *True* и внешний цикл прекращается. Если это неправда, то снова выполняется внутренний цикл. Он обрабатывает следующую строку данных (до `<eoln>`). Для нашего примера второе выполнение внутреннего цикла обработает вторую строку символов:

```
Он содержит две строки
```

После обработки второго `<eoln>` следующим символом будет `<eof>`, функция `End_Of_File` вернет *True,* и прекращается внешний цикл.

Для обработки произвольных текстовых файлов предусмотрены такие же процедуры, что и для стандартных файлов ввода-вывода. Единственное отличие — в список аргументов добавляется имя соответствующего внутреннего файла (указывается как первый аргумент).

Рассмотрим несколько примеров.

Если переменная `NextCh` имеет тип `Character`, то оператор вызова

```
Get ( Item => NextCh );
```

заносит следующий символ данных, набранный на клавиатуре, в переменную `Next`. В действительности это сокращенная форма следующего оператора:

```
Get ( File => Ada.Text_IO.Standard_Input, Item => NextCh );
```

Оператор

```
Get ( File => InData, Item => NextCh );
```

заносит следующий символ из файла `InData` в переменную `NextCh`. На этот «следующий» символ показывает указатель файла, который после каждого чтения автоматически передвигается.

Аналогичным образом операторы

```
Put ( Item => NextCh );
Put ( File => Ada.Text_IO.Standard_Input, Item => NextCh );
```

отображают значение `NextCh` на экране.

Оператор

```
Put ( File => OutData, Item => NextCh );
```

записывает значение `NextCh` в конец файла `OutData`.

Оператор `New_Line (OutData)` записывает в файл `OutData` маркер `<eoln>`.

Для текстовых файлов разрешены три режима работы:

```
In_File     -- только ввод из файла
            -- начало — с первого элемента
Out_File    -- только вывод в файл
            -- начало — с первого элемента
Append_File -- только вывод-добавление в файл
            -- начало — за последним элементом
```

Если операция обработки не соответствует режиму открытого файла, то возбуждается исключение `Mode_Error`. Элементы текстовых файлов могут иметь следующие типы: `Character`, `String`, числовые и перечисляемые типы.

Программа 18.1

Следующая программа копирует данные, набираемые на клавиатуре, в файл с именем `Data.txt`. Файловый объект `OutData` связывается с создаваемым файлом `Data.txt`, а затем используется для записи в этот файл. Если файл не может быть создан, генерируется исключение `Name_Error`.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main1 is
    OutData : Ada.Text_IO.File_Type;      -- внутренний файл
    File_Name : constant String := "Data.txt";       -- имя
    Ch : Character;                         -- читаемый символ
begin
    Create ( File => OutData, Mode => Out_File,
                Name => File_Name );
    while Not End_Of_File loop               -- для каждой строки
        while Not End_Of_Line loop           -- для каждого символа
            Get ( Ch ); Put ( OutData, Ch ); -- чтение/запись симв.
        end loop;
        Skip_Line; New_Line ( OutData );
                  -- след.строка / новая строка
    end loop;
    Close ( OutData );
    exception
        when Name_Error =>
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main1;
```

Программа 18.2

Следующая программа выводит на экран содержимое дискового файла `Data.txt` (который уже существует).

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main2 is
    InData : Ada.Text_IO.File_Type;               -- внутренний файл
    File_Name : constant String := "Data.txt";    -- имя
    Ch : Character;                               -- читаемый символ
begin
    Open ( File => InData, Mode => In_File,
                Name => File_Name );
    while Not End_Of_File ( InData ) loop
                       -- для каждой строки
        while Not End_Of_Line ( InData ) loop
                       -- для каждого символа
            Get ( InData, Ch ); Put ( Ch ); --чтение/вывод симв.
        end loop;
        Skip_Line ( InData ); New_Line;
                  -- след.строка/новая строка
    end loop;
    Close ( InData );
    exception
        when Name_Error =>
            Set_Col ( 10 );  -- сдвиг курсора в 10-ю колонку
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main2;
```

Программа 18.3

Следующая программа добавляет экземпляры целого типа `Numbers` в существующий дисковый файл `Data.txt`.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main3 is
    type Numbers is range 1 .. 100;
    AppData : Ada.Text_IO.File_Type;          -- внутренний файл
    File_Name : constant String := "Data.txt"; -- имя
    package Numbers_IO is new
                Ada.Text_IO.Integer_IO ( Numbers );
begin
    Open ( File => AppData, Mode => Append_File,
                Name => File_Name );
    for I in Numbers loop
        Numbers_IO.Put ( AppData, I );
        New_Line ( AppData );
    end loop;
    Close ( AppData );
    exception
        when Name_Error =>
            Put ("Не могу создать " & File_Name );
            New_Line;
end Main3;
```

ПРИМЕЧАНИЕ

- Процедура Skip_Line (<Имя>) применяется при чтении данных из дискового файла. Она перемещает «стрелку» указателя текстового файла в начало новой строки.
- Процедура New_Line (<Имя>) применяется при записи данных в дисковый файл. Она записывает в конец файла маркер <eoln>. Если между операторами вызова Put (<Имя>, Ch) пропущен оператор New_Line (<Имя>), то в файле будет отсутствовать разбивка на строки, то есть данные файла образуют одну длинную строку.
- Если, например, в программе 18.1 пропустить оператор Skip_Line, то после записи в дисковый файл одной строки в него будут записываться только маркеры <eoln>.

Программа 18.4

Разработаем пакет для ввода/вывода календарных дат. Создадим его на основе стандартного пакета службы времени `Ada.Calendar`. Пакет `Ada.Calendar` содержит определение приватного типа `Time`. Текущее значение переменной этого типа возвращает функция `Clock`. В пакете имеются функции `Year`, `Month`, `Day`. Они возвращают числовые значения типов `Year_Number`, `Month_Number`, `Day_Number`.

```
with Ada.Text_IO, Ada.Calendar;
use Ada.Text_IO;
package Calendar_Dates is
    type Months is ( Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec );
    type Date is
        record
            Day : Ada.Calendar.Day_Number;
```

```
            Month: Months;
            Year : Ada.Calendar.Year_Number;
        end record;
    procedure Get ( Item : out Date );
    procedure Put ( Item : in Date );
    function Today return Date;
end Calendar_Dates;
with Ada.Text_IO, Ada.Calendar, Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
package body Calendar_Dates is
    package Month_IO is new
                    Ada.Text_IO.Enumeration_IO ( Months );
    procedure Get ( Item : out Date ) is
    begin      -- вводит дату в формате DD MMM YYYY
        Get ( Item.Day );
        Month_IO.Get ( Item.Month );
        Get ( Item.Year );
    end Get;
    procedure Put ( Item : in Date ) is
    begin     -- отображает дату в формате DD MMM YYYY
        Put ( Item.Day, width => 1 );
        Ada.Text_IO.Put (' ');
        Month_IO.Put ( Item.Month, width => 1 );
        Ada.Text_IO.Put (' ');
        Put ( Item.Year, width => 4 );
    end Put;
    function Today return Date is
                    -- возвращает текущую дату
        Now : Ada.Calendar.Time;
        Tmp : Date;
    begin
        Now := Ada.Calendar.Clock; -- получение кода времени
        Tmp.Day := Ada.Calendar.Day ( Now );
                -- извлечение дня из кода времени
        Tmp.Month := Months'Val (Ada.Calendar.Month (Now)-1);
        Tmp.Year := Ada.Calendar.Year ( Now );
                -- извлечение года из кода времени
        return Tmp;
    end Today;
end Calendar_Dates;
```

Клиентом этого пакета может быть следующая программа:

```
with Ada.Text_IO, Calendar_Dates, Ada.Integer_Text_IO;
use Ada.Integer_Text_IO, Ada.Text_IO, Calendar_Dates;
procedure Test_Dates is
    D : Date;
begin
    D := Today;
    Put (" Today is ");
    Put ( D );    -- процедура из пакета Calendar_Dates;
    New_Line;
    Put (" Please, enter a date (DD MMM YYYY)> ");
    Get ( D );    -- процедура из пакета Calendar_Dates;
    New_Line;
    Put (" You entered: ");
    Put ( D );
end Test_Dates;
```

Двоичные файлы последовательного доступа

Файл последовательного доступа — это структура данных, состоящая из линейной последовательности компонентов одного типа, а также имеющая переменную длину (без ограничения на максимальный размер).

В отличие от текстовых файлов, двоичные файлы не могут непосредственно читаться человеком. Элементы двоичных файлов хранятся не в виде символов, а в том же формате, который принят для представления элементов данных в оперативной памяти. Например, элементы типа `Integer` запоминаются в виде 16-разрядных двоичных слов, элементы типа `Float` — в виде 32-разрядных двоичных слов. Иначе говоря, двоичные файлы можно считать фрагментами внутренней памяти, размещенными во внешней памяти.

Средства для работы с двоичными файлами последовательного доступа находятся в родовом пакете `Ada.Sequential_IO`.

Последовательные двоичные файлы состоят из элементов типа `Element_Type`, который является формальным родовым параметром пакета:

```
generic
    type Element_Type ( <> ) is private;
package Ada.Sequential_IO is
    ...
```

Из данного объявления следует, что элементы двоичного последовательного файла могут иметь числовой или перечисляемый тип, могут быть массивами и записями.

Разбивки на строки в двоичных файлах нет, поэтому маркер `<eoln>` не используется, а в пакете `Ada.Sequential_IO` отсутствует функция `End_Of_Line`.

Для работы с последовательными двоичными файлами разрешены три режима: `In_File`, `Out_File`, `Append_File`. Для ввода-вывода элементов файла используются процедуры `Read`, `Write`.

Процедура `Read` имеет следующую спецификацию:

```
procedure Read ( File : in File_Type;
            Item : out Element_Type );
```

Эта процедура читает из заданного файла один элемент и заносит его значение в параметр `Item`. При этом предполагается, что дисковый файл открыт для работы в режиме `In_File`. После выполнения процедуры «стрелка» указателя файла перемещается на следующий элемент.

Спецификация процедуры `Write` записывается в виде:

```
procedure Write ( File : in File_Type;
            Item : in Element_Type );
```

Эта процедура записывает в файл (после последнего элемента) значение нового элемента. При этом предполагается, что дисковый файл открыт для работы в режиме `Out_File` или `Append_File`.

В качестве примера рассмотрим программу, в которой:

1) создается экземпляр пакета `Ada.Sequential_IO` для работы с типом `Person`;
2) создается двоичный дисковый файл `People.txt`. Для него определяется режим работы `Out_File` (вывод), задается связывание с внутренним файлом `Data`;

3) в двоичный внешний файл `People.txt` записываются двоичные образы объектов типа `Person`;
4) меняется режим работы дискового файла. Задается режим ввода (`In_File`). Стрелка указателя файла перемещается на первый элемент;
5) читаются данные из дискового файла. Их содержимое выводится на экран;
6) внешний файл закрывается;
7) для обеспечения определения в отношении исключения `Name_Error` используется пакет `Ada.Text_IO`.

Программа 18.5

```
with Ada.Text_IO; Ada.Sequential_IO;
use Ada.Text_IO;
procedure Main4 is
    type T_Sex is ( Female, Male );
    subtype T_Height is Integer range 0 .. 250;
    type Person is record
            Name : String ( 1 .. 6 );
            Height : T_Height := 0;  -- рост в см.
            Sex : T_Sex;
        end record;
    procedure Output_Person ( One_Person : in Person ) is
    begin    -- начало процедуры
        Put ( One_Person.Name );
        Put ( " is " );
        Put ( Integer'Image ( One_Person.Height ));
        Put ("cm and is ");
        if One_Person.Sex = Female then Put ( "Female" );
                        else Put ( "Male" );
        end if;
        New_Line;
    end Output_Person;    -- конец процедуры
    type Person_Index is range 1 .. 2;
    subtype Person_Range is Person_Index;
    type Person_Array is array ( Person_Range ) of Person;
    File_Name : constant String := "People.txt";
    People : Person_Array;
    package Person_IO is new Ada.Sequential_IO ( Person );
        Data : Person_IO.File_Type;  -- внутр.файл
    begin   -- начало программы
        People ( 1 ) := ( Name => "Peter ", Height => 180,
                    Sex => Male );
        People ( 2 ) := ( Name => "Eva   ", Height => 165,
                    Sex => Female );
        Person_IO.Create ( File => Data,
                Mode => Person_IO.Out_File, Name => File_Name );
        for i in Person_Range loop
            Person_IO.Write ( Data, People ( i ));
        end loop;
        Reset ( File => Data, Mode => Person_IO.In_File );
        for i in Person_Range loop
            Person_IO.Read ( Data, People ( i ));
            Output_Person ( People ( i ));
        end loop;
        Person_IO.Close ( Data );
    exception
```

продолжение ⇩

```
        when Name_Error =>
            Put ( "Can not create " & File_Name );
            New_Line;
end Main4;     -- конец программы
```

Двоичные файлы прямого доступа

Организация *файла прямого доступа* делает возможным доступ к произвольному компоненту файла. Индекс, используемый для доступа к компоненту, обычно называется *ключом*. Если ключ реализован в виде целого числа, то он очень похож на обычный индекс, используемый для обозначения компонентов массива. Однако в целом реализация файла прямого доступа и операций выбора его компонентов сильно отличается от реализации массива, так как файлы хранятся не в основной памяти, а на внешних запоминающих устройствах.

Средства для работы с двоичными файлами прямого доступа находятся в родовом пакете `Ada.Direct_IO`. Элементы этих файлов также должны принадлежать к типу `Element_Type`, который является формальным родовым параметром пакета.

Для работы с файлами прямого доступа разрешены три режима: `In_File` (ввод из файла), `InOut_File` (ввод из файла и вывод в файл), `Out_File` (вывод в файл).

Основное отличие файлов прямого доступа — обработка их элементов производится в произвольном порядке. На практике для работы с элементом такого файла надо указывать его индекс. Индекс отображает положение элемента в файле. Непосредственно после открытия файла значение индекса устанавливается равным 1.

Для обработки индекса в пакете `Ada.Direct_IO` имеются:

- процедура установки индекса `Set_Index`;
- функция определения индекса `Index`;
- функция определения максимального индекса `Size`.

Процедура

```
procedure Set_Index ( File : File_Type; To : Positive_Count );
```

устанавливает индекс, равный значению параметра `To`.

ПРИМЕЧАНИЕ

Диапазон значений подтипа Positive_Count — от 1 до зависящего от реализации числа Count'Last.

Функция

```
function Index ( File : File_Type ) return Positive_Count;
```

возвращает значение, равное текущему индексу файла.

Имеются две модификации процедуры чтения элемента из файла прямого доступа и занесения его в переменную `Item`:

```
procedure Read ( File : in File_Type; Item : out Element_Type;
                 From : in Positive_Count );
procedure Read ( File : in File_Type; Item : out Element_Type);
```

Для первой модификации из файла читается элемент, индекс которого указывается параметром `From`. Для второй модификации индекс считываемого элемента равен текущему значению индекса файла. После выполнения операции ввода значение индекса увеличивается на единицу.

Аналогичным образом предусмотрены две разновидности процедуры записи из переменной `Item` в элемент файла:

```
procedure Write ( File : in File_Type; Item : in Element_Type;
           To : in Positive_Count );
procedure Write ( File : in File_Type; Item : in Element_Type );
```

Параметр `To` задает индекс элемента, в который записываются данные. Если параметр `To` не указан, то используется текущее значение индекса файла. После завершения вывода значение индекса увеличивается на 1.

Максимально возможное значение индекса файла возвращается с помощью функции

```
function Size ( File : File_Type ) return Count;
```

В качестве примера рассмотрим программу, в которой:

1) создан экземпляр пакета `Ada.Direct_IO` — для работы с типом `Book`;
2) объявлена процедура `Format` — для создания дискового файла прямого доступа (каталога книг) и последовательного заполнения его фиктивным содержимым — пробелами;
3) объявлена процедура `Load` — для загрузки в каталог фактической информации (по конкретному индексу — номеру);
4) объявлена процедура `View` — для просмотра на дисплее записи с конкретным номером;
5) приведены примеры использования этих процедур для решения конкретных задач.

Программа 18.6

```
with Ada.Text_IO, Ada.Direct_IO;
use Ada.Text_IO;
procedure Main5 is
    type Book is record
        Number : Positive;
        Author : String ( 1 .. 17 ) := ( others => ' ' );
        Name : String ( 1 .. 25 ) := ( others => ' ' );
    end record;
    package Book_IO is new Ada.Direct_IO ( Book );
    One_Book : Book;
    procedure Format ( N : Positive; Cat_Name : String ) is
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
        Book_IO.Create ( File => Inner,
                    Mode => Book_IO.Out_File,
                    Name => Cat_Name );
        for i in 1 .. N loop
            One_Book.Number := i;
            Book_IO.Write ( Inner, One_Book );
        end loop;
        Book_IO.Close ( Inner );
```

продолжение ⇩

```
    end Format;
    procedure Load ( One : Book; I : Book_IO.Positive_Count;
                Cat_Name : String ) is
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
        Book_IO.Open ( File => Inner,
                    Mode => Book_IO.Out_File,
                    Name => Cat_Name );
        Book_IO.Write ( Inner, One, I );
        Book_IO.Close ( Inner );
    end Load;
    procedure View ( I : Book_IO.Positive_Count;
                Cat_Name : String ) is
        One : Book;
        Inner : Book_IO.File_Type;  -- внутренний файл
    begin
        Book_IO.Open ( File => Inner,
                    Mode => Book_IO.In_File,
                    Name => Cat_Name );
        Book_IO.Read ( Inner, One, I );
        Put ("Number is "); Put ( Positive'Image (One.Number));
        New_Line;
        Put ("Author is "); Put_Line ( One.Author );
        Put ("Name is "); Put_Line ( One.Name );
        Book_IO.Close ( Inner );
    end View;
    File_Name : constant String := "Catalog.txt";
begin     -- начало программы
    Format ( 10_000, File_Name ); -- форматирование каталога
    One_Book.Number := 10;
    One_Book.Author := "Н.Гоголь";
    One_Book.Name := "Ревизор";
    Load ( One_Book, 10, File_Name );   -- запись в каталог
    One_Book.Number := 20;
    One_Book.Author := "М.Лермонтов";
    One_Book.Name := "Демон";
    Load ( One_Book, 20, File_Name );   -- запись в каталог
    View ( 20, File_Name );  -- просмотр 20-й записи
end Main5;    -- конец программы
```

Потоки ввода-вывода

Пакеты последовательного и прямого ввода-вывода обрабатывают только однородные файлы (все элементы файла имеют один и тот же тип). Потоковый ввод-вывод позволяет обрабатывать неоднородные файлы. В языке Ada поток — это последовательность элементов, составленных из величин различных типов.

Основная идея — с любым файлом, объявленным с использованием пакета `Ada.Streams.Stream_IO`, связывается поток. Такой файл может быть обработан последовательно (с использованием потокового механизма) или с указанием позиции (как в пакете прямого ввода-вывода). Пакет `Ada.Streams.Stream_IO` позволяет создавать, открывать и закрывать файл. Дополнительно здесь имеется функция `Stream`, которая для заданного потокового файла возвращает ссылку на поток, связанный с файлом.

Спецификация этой функции имеет вид:

```
function Stream ( File : in File_Type ) return Stream_Access;
```

ПРИМЕЧАНИЕ

Все потоки являются производными от абстрактного типа Streams.Root_Stream_Type. Доступ к потоку осуществляется с помощью параметра-ссылки, указывающего на объект типа Streams.Root_Stream_Type'Class.

Последовательная обработка потоков выполняется с помощью атрибутных функций: `T'Read`, `T'Write`, `T'Input`, `T'Output`. Эти атрибуты определены для всех нелимитированных типов. Пользователь может заменить их, предусматривая спецификатор определения атрибута, может явно определить такие атрибуты для лимитированных типов. Параметры атрибутов `Read` и `Write` обозначают поток и элемент типа `T`:

```
procedure T'Write ( Stream : access
            Streams.Root_Stream_Type'Class;
        Item : in T );
procedure T'Read ( Stream : access
            Streams.Root_Stream_Type'Class;
        Item : out T );
```

Рассмотрим пример. Предположим, что надо записать смесь из целых, имен месяцев и дат, причем тип `Date` имеет вид

```
type Date is
    record
        Day : Integer;
        Month : Month_Name;
        Year : Integer;
    end record;
```

Решение

1. Обычным способом создается файл и получается доступ к связанному потоку:

```
use Streams. Stream_IO;
Mixed_File : File_Type;  -- внутренний файл
S : Stream_Access;       -- ссылочная переменная;
...
Create ( Mixed_File, "Data.doc" );
        -- создаем файл для хранения смеси
S := Stream ( Mixed_File );
        -- получаем ссылку на внутренний файл
```

2. Вызываются атрибуты для записи величин в поток:

```
Date'Write ( S, Some_Date );
Integer'Write ( S, Some_Integer );
Month_Name'Write ( S, This_Month );
...
```

ПРИМЕЧАНИЕ

Streams.Stream_IO — это не родовой пакет, он не должен конкретизироваться.

3. Все неоднородные файлы имеют один и тот же тип. Все они являются двоичными файлами.
4. Для чтения файла, записанного таким образом, должна использоваться симметричная последовательность атрибутов:

```
Date'Read ( ...
Integer'Read ( ...
Month_Name'Read ( ...
```

Если для чтения используются неподходящие подпрограммы, то будет получено странное значение или исключение `Data_Error`.

В случае простой записи, такой как `Date`, предопределенный атрибут `Write` просто вызывает атрибуты для ее компонентов. Иными словами, реализация этого атрибута имеет вид:

```
procedure Date'Write ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in Date ) is
begin
    Integer'Write ( Stream, Item.Day );
    Month_Name'Write ( Stream, Item.Month );
    Integer'Write ( Stream, Item.Year );
end Date'Write;
```

Мы можем применить собственный вариант атрибута `Write`. Положим, что имя месяца в дате надо выводить как целое число. Тогда мы должны записать другую процедуру:

```
procedure Date_Write ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in Date ) is
begin
    Integer'Write ( Stream, Item.Day );
    Integer'Write ( Stream, Month_Name'Pos (Item.Month) + 1 );
    Integer'Write ( Stream, Item.Year );
end Date_Write;
```

Для замены стандартной процедуры на построенную нам необходимо записать:

```
for Date'Write use Date_Write;
```

В этом случае оператор вызова `Date'Write ( S, Some_Date )` будет использовать новый формат для вывода дат.

Подобные возможности применимы и к вводу. Иначе говоря, мы должны будем объявить полную версию `Date'Read` для чтения месяца как целого числа и преобразования в соответствующее значение типа `Month_Name`.

ПРИМЕЧАНИЕ

- Мы изменяли формат вывода месяца только в датах. Если формат вывода месяца изменяется во всех случаях, то вместо переопределения `Date'Write` мы бы переопределили `Month_Name'Write`. Это оказало бы косвенное воздействие на изменение вывода дат.
- Предопределенные атрибуты `T'Read` и `T'Write` могут быть перекрыты только в той спецификации пакета, где объявлен тип `T`.

СЛЕДСТВИЕ

Для предопределенных типов эти атрибуты не могут быть изменены. Но они могут изменяться для типов, производных от них.

Ввод-вывод усложняется в случае работы с массивами и записями с дискриминантами, так как необходимо учесть закрытую информацию, представленную границами и дискриминантами. (В случае дискриминанта со значением по умолчанию он обрабатывается как обычный компонент.)

Учет закрытой информации выполняется при помощи дополнительных атрибутов `'Input` и `'Output`.

Основная идея — атрибуты обрабатывают закрытую информацию (если она есть), а затем вызывают процедуры `Read` и `Write` для обработки остатка значения.

Они имеют следующие спецификации:

```
procedure T'Output ( Stream : access
            Streams.Root_Stream_Type'Class;
            Item : in T );
function T'Input ( Stream : access
    Streams.Root_Stream_Type'Class ) return T;
```

Отметим, что `Input` реализован в виде функции, так как тип `T` может быть неопределенным и мы можем не знать ограничения при конкретном вызове.

В случае массива процедура `Output` выводит границы массива, а затем вызывает атрибут `Write` для вывода самого значения.

В случае комбинированного типа:

- Если дискриминант определен, то процедура `Output` просто вызывает атрибут `Write` (он обрабатывает дискриминант так же, как и другие компоненты).
- Если дискриминант не определен, то процедура `Output` вначале выводит дискриминант, а затем вызывает атрибут `Write` для обработки остатка записи.

Например, если задан подтип

```
subtype String_6 is String ( 1 .. 6 );
S : String_6 := "String";
```

то можно записать:

```
String_6'Output ( S );
            -- в файл выводятся и границы массива;
String_6'Write ( S ); -- границы в файл не выводятся.
```

Приведенное выше описание `T'Input`, `T'Output` относится к подразумеваемым атрибутам. Они могут быть переопределены для выполнения любых действий так, что не потребуется вызывать `T'Read`, `T'Write`. Более того, `Input` и `Output` существуют и для определенных подтипов — по умолчанию они просто вызывают `Read` и `Write`.

Для работы с надклассовыми типами имеются атрибуты: `T'Class'Output` и `T'Class'Input`.

При выводе: выводится внешнее представление тега, а затем вызывается процедура `Output` для конкретного типа (используется механизм диспетчеризации). Эта процедура выводит конкретное значение (вызовом процедуры `Write`).

Для ввода: вначале читается тег, а затем в соответствии с его значением вызывается соответствующая функция `Input` (диспетчеризацией).

Определены также атрибуты `T'Class'Write` и `T'Class'Read`. Они выполняют диспетчеризацию атрибутных подпрограмм `Write` и `Read` для конкретного типа на основе идентификации тега.

Основной принцип — все, что записано, может быть обратно прочитано с помощью соответствующей обратной операции.

Вернемся к рассмотрению основной структуры. Все потоки являются производными от абстрактного типа `Streams.Root_Stream_Type`, который имеет две абстрактные операции `Read` и `Write`:

```
procedure Read ( Stream : in out Root_Stream_Type;
            Item : out Stream_Element_Array;
            Last : out Stream_Element_Offset ) is abstract;
procedure Write ( Stream : in out Root_Stream_Type;
            Item : in Stream_Element_Array ) is abstract;
```

Они работают в терминах потоковых элементов, а не отдельных типизированных величин.

Предопределенные атрибуты `Read` и `Write` используют операции `Read` и `Write` ассоциированного потока. Пользователь может переопределить содержание атрибутов. Заметим, что в корневом типе параметр `Stream` принадлежит типу `Root_Stream_Type`, тогда как такой же параметр производного атрибута имеет ссылочный тип, обозначающий соответствующий класс. Таким образом, любой пользовательский атрибут будет выполнять разыменование (снятие косвенности):

```
procedure My_Write ( Stream : access
                Streams.Root_Stream_Type'Class; Item : T ) is
begin
        -- преобразование величины в потоковые элементы
    Streams.Write ( Stream.all, ... );  -- диспетчирование
end My_Write;
```

Пакет `Stream_IO` может использоваться и для индексированного доступа. Это возможно, так как файл является последовательностью потоковых элементов. Индексация для потоковых элементов работает так же, как и для типизированных элементов в пакете `Direct_IO`. Индекс может быть прочитан и сброшен. Предусмотрены варианты процедур `Read` и `Write` как для последовательной, так и для индексной обработки.

Объектно-ориентированный ввод-вывод в языке C++

В языке C++ основным механизмом ввода-вывода является поток: последовательность байтов (символов), которая с точки зрения программы не зависит от конкретных устройств — источников или приемников обмениваемых данных. Этими устройствами могут быть файл на диске, принтер, клавиатура, дисплей и т. д. При обмене с потоком используется вспомогательный участок основной памяти — буфер потока (буфер вывода или буфер ввода).

В буфер потока помещаются выводимые программой данные перед тем, как они будут переданы внешнему устройству. При вводе данных они вначале заносятся в буфер и лишь затем передаются в область памяти выполняемой программы. Использование буфера как промежуточной ступени при обменах с устройствами ввода-вывода повышает скорость передачи данных, поскольку реальные пересылки осуществляются при заполнении (на выводе) или опустошении буфера (на вводе).

Большую часть работы, связанной с заполнением и очисткой буферов ввода-вывода, берет на себя операционная система и выполняет ее без явного участия программиста. Поэтому поток в программе можно рассматривать просто как последовательность байтов. Важно отметить, что никакой связи значений этих байтов с кодами определенного алфавита не предусматривается. Задача программиста при вводе-выводе с помощью потоков — установить соответствие между объектами-участниками обмена и последовательностью байтов потока, в которой могут отсутствовать сведения о типах представляемых (передаваемых) данных.

Используемые в программах потоки делятся на три категории:

- входные, из которых читаются данные;
- выходные, в которые вводятся данные;
- двунаправленные, допускающие как чтение, так и запись.

Все потоки ввода-вывода имеют последовательную структуру, то есть в каждый момент для потока определены позиции записи и/или чтения, и эти позиции после обмена перемещаются по потоку на длину переданной порции данных.

В соответствии с особенностями устройства ввода-вывода, к которому «присоединен» поток, различают стандартные, строковые и файловые потоки.

Стандартные потоки соответствуют передаче данных от клавиатуры и к экрану дисплея. Стандартным устройством ввода считается клавиатура, а стандартным устройством вывода — экран дисплея.

Строковый поток состоит из символов, в совокупности образующих символьный массив (строку) в основной памяти.

Файловый поток несет в себе символы, которые размещаются на внешнем носителе данных компьютера (например, на магнитном или оптическом диске).

Обсудим принципиальные особенности потоковой библиотеки ввода-вывода языка C++.

Потоковая библиотека ввода-вывода

В библиотеке определено целое семейство классов, которые обеспечивают ввод и вывод данных на стандартные устройства и в файлы. Дополнительные классы организуют форматирование текста в оперативной памяти, поддерживая «файловое» представление строк. Каждый из этих классов ввода-вывода задает, как осуществляется обмен значениями встроенных типов данных. Кроме того, средства библиотеки используются для ввода-вывода данных в объекты пользовательских классов.

Базовыми библиотечными средствами ввода-вывода являются:

- Класс `istream` (input stream — поток ввода) обеспечивает операции ввода.
- Класс `ostream` (output stream — поток вывода) обеспечивает операции вывода.

- Объект `cin` класса `istream` читает данные со стандартного устройства ввода.
- Объект `cout` класса `ostream` записывает данные на стандартное устройство вывода.
- Объект `cerr` класса `ostream` записывает данные на стандартное устройство сообщений об ошибке. Объект `cerr`, как правило, используется для сообщений об ошибках в программе.
- Операция *извлечения* `>>` используется для чтения данных, передаваемых в объект класса `istream`. Она извлекает данные из потокового объекта, стоящего слева, и присваивает эти данные переменной, стоящей справа.
- Операция *вставки* `<<` используется для записи данных, передаваемых в объект класса `ostream`. Она копирует содержимое переменной, стоящей справа, в объект, содержащийся слева.

Концептуально ни вид устройства, ни специфика набора символов не влияет на выполнение операций ввода-вывода. В частности, операцию `>>` можно использовать для чтения данных независимо от того, осуществляется ли чтение с консоли, из файла на диске или из строки в оперативной памяти. Кроме того, применение операции не зависит от типа читаемых символов.

Классы ввода-вывода определены в трех отдельных заголовочных файлах библиотеки: в `iostream.h` определены классы для чтения и записи данных на стандартные устройства, в `fstream.h` определены классы для чтения и записи данных в именованный файл, а в `sstream.h` определены классы для чтения и записи данных в строку, расположенную в оперативной памяти. Каждый из классов, перечисленных в заголовочных файлах `fstream` и `sstream`, является наследником соответствующего класса, указанного в заголовочном файле `iostream`. Список классов ввода-вывода содержится в табл. 18.5.

Таблица 18.5. Заголовки и классы библиотеки ввода-вывода

Заголовочный файл	Класс
iostream.h	istream — читает данные из потока ostream — записывает данные в поток iostream — читает и записывает данные в поток. Наследник классов istream и ostream
fstream.h	ifstream — читает данные из файла. Наследник класса istream ofstream — записывает данные в файл. Наследник класса ostream fstream — читает и записывает данные в файл. Наследник класса iostream
sstream.h	istringstream — читает данные из строки. Наследник класса istream ostringstream — записывает данные в строку. Наследник класса ostream stringstream — читает и записывает данные в строку. Наследник класса iostream

Отношения наследования между потоковыми классами позволяют сделать следующие выводы. Каждая программа, в которой объекты класса `istream` применялись для чтения со стандартного устройства, может быть переписана для чтения данных из файла (при помощи класса `ifstream`) или строки (при помощи класса `istringstream`). Аналогично, в программах для вывода данных можно использовать классы `ofstream` или `ostringstream` вместо класса `ostream`. Класс `iostream` является наследником классов `istream` и `ostream`. Это значит, что объект класса `iostream` владеет операциями обоих родительских классов. Иначе говоря, объект класса `iostream` можно использовать как для ввода, так и для вывода данных в тот же поток. В библиотеке определены также еще два наследника от класса `iostream` — классы `fstream` и `stringstream`. Эти классы предназначены для чтения и записи данных в файл или строку.

Как известно, функция, обрабатывающая указатели на объекты базового класса, может работать с объектами производного класса. Отсюда следует, что функция с параметром типа `istream&` может быть вызвана для объектов класса `ifstream` или `istringstream`. Аналогично, функция с параметром типа `ostream&` может быть вызвана для объектов класса `ofstream` или `ostringstream`. Так как классы ввода-вывода связаны наследованием, можно написать одну функцию и применять ее ко всем трем видам потоков: стандартному, файловому или строковому потоку.

Стандартные потоки

Рассмотрим пример программы с использованием стандартных потоков. Пользователю предлагается ввести радиус области видимости, а программа подсчитывает и выводит на экран площадь области, ограниченной данным радиусом.

```
// scope.cpp
//работа со стандартными потоками
#include <iostream>    //для cout и т. д.
using namespace std;
int main()
{
  float radius;          // переменная вещественного типа
  const float Pi =3.14;  // вещественная константа
  cout << "Введите радиус области видимости: ";// запрос
  cin >> radius;    // получение радиуса
  float area = Pi * radius * radius;    // вычисление площади области видимости
  cout << "Площадь области видимости равна " << area << endl;// вывод результата
                                                              // на экран
  return 0;
}
```

Первые две строки, с которых начинается программа `scope`, являются *директивами*. Первая строка представляет собой *директиву препроцессора*, вторая — директиву `using`. Директивы нельзя считать частью языка C++, однако их использование является необходимым.

Первая строка программы `#include <iostream>` похожа на оператор, но это на самом деле не так. Она не входит в тело какой-либо из функций и не заканчивается символом «;», как операторы C++. Кроме того, эта строка начинается с символа #.

Такая строка называется *директивой препроцессора*. Напомним, что любой оператор является указанием *компьютеру* совершить какое-либо действие. Директива препроцессора, напротив, является указанием *компилятору*. Препроцессором называется специальная часть компилятора, обрабатывающая подобные директивы перед началом процесса компиляции кода.

Директива `#include` предлагает препроцессору включить в компилируемый файл содержимое другого файла. Это содержимое подставляется на место директивы `#include`. Директива `#include` действует так же, как действует пользователь, копируя необходимый текст в текстовом редакторе, а затем вставляя его в нужное место документа. Директива `#include` является одной из многих директив препроцессора, каждая из которых предваряется символом `#`. Файл, включаемый с помощью директивы `#include`, обычно называют *заголовочным файлом*.

В программе `scope` директива `#include` является указанием включить в исходный текст содержимое файла `iostream` перед компиляцией кода. Имя `iostream` является примером имени заголовочного (включаемого) файла. Файл `iostream` содержит описания, необходимые для работы с объектами `cout` и `cin`, а таже операциями `<<` и `>>`. Без этих описаний компилятору не будет известно, что значат имена `cout` и `cin`, а употребление операций `<<` и `>>` будет воспринято как некорректное. Существует множество заголовочных файлов. В стандартном C++ заголовочные файлы не имеют расширения, но те файлы, которые были унаследованы от языка C, имеют расширение `.h`.

В общем случае области действия всех имен для программы на языке C++ можно разбить на несколько так называемых *пространств имен*. *Пространством имен* называется область действия программы, в которой распознается определенная совокупность имен. Эти имена неизвестны за пределами данного пространства имен. Директива

```
using namespace std;
```

означает, что все определенные ниже имена в программе будут относиться к пространству имен с именем `std`. Различные элементы программы описаны с использованием пространства имен `std`, например объекты `cout` и `cin`. Если не использовать директиву `using`, то к этим элементам программы придется каждый раз добавлять имя `std`:

```
std::cout << "Введите радиус области видимости: ";
```

Для того чтобы не дописывать `std::` каждый раз перед именем объекта, используется директива `using`.

Оператор нашей программы

```
cout << "Введите радиус области видимости: ";
```

выводит на экран строку, заключенную в кавычки.

Здесь имя `cout` обозначает объект C++, предназначенный для работы со *стандартным потоком вывода*. Напомним, что поток — это некоторая абстракция, отражающая перемещение данных от источника к приемнику. Стандартный поток вывода обычно направлен на экран, хотя допускается возможность его перенаправления на другие устройства вывода.

Операция *вставки* `<<` (вывода) копирует содержимое правого операнда в объект, являющийся ее левым операндом. В программе `scope` операция `<<` направляет

строку "Введите радиус области видимости: " в объект cout, который выводит эту строку на экран.

Заметим, что запись потокового вывода чрезвычайно проста, а операция << и объект cout знают, каким образом отличать строку от целого числа и как обрабатывать каждое из них. Если мы выводим на экран целое число, то оно отображается в числовом формате. Если мы выводим строку, то она отображается в виде текста. Это кажется очевидным, но за подобными действиями стоит механизм переопределения (перегрузки) операций, типичный для C++.

Оператор

```
cin >> radius;
```

заставляет программу ожидать ввода числа от пользователя. Введенное значение присваивается переменной radius. Имя cin указывает на объект, определенный в C++ для работы со *стандартным потоком ввода*. Этот поток содержит данные, вводимые с клавиатуры (если он не переопределен). Операция *извлечения* >> (ввода) изымает данные из потокового объекта, являющегося ее левым операндом, и присваивает эти данные правому операнду — переменной.

Следующий оператор программы вычисляет значение площади видимости, за ним следует вывод результата на экран с помощью стандартного потока вывода.

Обсудим конструкцию

```
cout << "Площадь области видимости равна " << area << endl;
```

более подробно.

Здесь операция << повторяется несколько раз, или *каскадируется*. Такая конструкция вполне легальна, потому что в этом случае операция << сначала посылает в объект cout строку "Площадь области видимости равна ", затем значение переменной area и, наконец, манипулятор endl. Манипулятор endl означает вставку в символьный поток символа окончания строки, поэтому весь последующий текст будет печататься с новой строки. Строго говоря, манипулятор endl связан с очисткой буфера выходного потока. Манипулятор — это особая функция, обращенная к потоку и предназначенная для изменения вывода. Существуют и другие манипуляторы.

Аналогичным путем можно каскадировать и операцию извлечения >> (совместно с объектом cin), давая возможность пользователю вводить несколько значений подряд. Однако при этом теряется возможность формирования приглашения для каждого ввода пользователя.

Форматирование потоков

Корнем в библиотеке всех потоковых классов является класс ios. Он определяет такие возможности, как флаги форматирования, флаги ошибок и режимы работы с файлами. Обсудим флаги форматирования, а затем флаги состояния ошибки. Режимы работы с файлами опишем при рассмотрении файловых потоков.

Флаги форматирования

Флаги форматирования включают в себя набор определений перечисляемого типа в классе ios. Флаги являются переключателями, определяющими различные форматы и способы ввода-вывода (табл. 18.6).

Таблица 18.6. Флаги форматирования класса ios

Флаг	Значение
skipws	Пропуск пробелов при вводе
left	Выравнивание по левому краю
right	Выравнивание по правому краю
internal	Заполнение между знаком или основанием числа и самим числом
dec	Перевод в десятичную форму
oct	Перевод в восьмеричную форму
hex	Перевод в шестнадцатеричную форму
boolalpha	Перевод логического 0 и 1 соответственно в "false" и "true"
showbase	Выводить индикатор основания системы счисления (0 для восьмеричной, 0x для шестнадцатеричной)
showpoint	Показывать десятичную точку при выводе
uppercase	Переводить в верхний регистр буквы X, E и буквы шестнадцатеричной системы счисления (ABCDEF) (по умолчанию — в нижнем регистре)
showpos	Показывать «+» перед положительными целыми числами
scientific	Экспоненциальный вывод чисел с плавающей запятой
fixed	Фиксированный вывод чисел с плавающей запятой
unitbuf	Сброс потоков после вставки
stdio	Сброс stdout, sterror после вставки

Обращаются к флагам форматирования посредством указания имени класса и оператора явного задания флага (например, ios::skipws). Любые флаги могут быть установлены с помощью методов setf() и unsetf(). Приведем примеры:

```
cout.setf(ios::left);    //выравнивание текста по левому краю
cout >> "Этот текст выровнен по левому краю"
cout.unsetf(ios::left);  //вернуться к прежнему форматированию
```

Многие флаги можно выставить с помощью манипуляторов.

Манипуляторы

Манипуляторы — это функции форматирования, вставляемые прямо в поток. Например, манипулятор endl посылает символ разделителя строк в поток и сбрасывает буфер:

```
cout << "Все хорошо, прекрасная маркиза!" << endl;
```

Различают манипуляторы без аргументов (табл. 18.7) и с аргументами (табл. 18.8).

Таблица 18.7. Манипуляторы ios без аргументов

Манипулятор	Назначение
ws	Включает пропуск пробелов при вводе
dec	Перевод в десятичную форму
oct	Перевод в восьмеричную форму
hex	Перевод в шестнадцатеричную форму
endl	Вставка разделителя строк и очистка выходного потока
ends	Вставка символа отсутствия информации для окончания выходной строки

Манипулятор	Назначение
flush	Очистка выходного потока
lock	Закрыть дескриптор файла
unlock	Открыть дескриптор файла

Вставляются эти манипуляторы прямо в поток. Например, для вывода переменной var в шестнадцатеричной форме надо записать:

```
cout << hex << var;
```

Доступ к этим возможностям обеспечивает заголовочный файл iomanip.

Таблица 18.8. Манипуляторы ios с аргументами

Манипулятор	Аргумент	Назначение
setw()	Ширина поля (int)	Устанавливает ширину поля для вывода данных
setfill()	Символ заполнения (int)	Устанавливает символ заполнения (по умолчанию пробел)
setprecision()	Точность (int)	Устанавливает точность (число выводимых знаков)
setiosflags()	Флаги форматирования (long)	Устанавливает указанные флаги форматирования
resetiosflags()	Флаги форматирования (long)	Сбрасывает указанные флаги форматирования

ВНИМАНИЕ

Манипуляторы действуют лишь на те данные, которые следуют за ними в потоке, а не на те, которые находятся перед ними.

Функции

Класс ios содержит набор функций для установки флагов форматирования и выполнения других действий (табл. 18.9).

Таблица 18.9. Основные функции ios

Функция	Назначение
ch = fill ();	Возвращает символ заполнения (символ, которым заполняется неиспользуемая часть текстового поля; по умолчанию пробел)
fill (ch);	Устанавливает символ заполнения
p = precision();	Возвращает значение точности (число выводимых знаков для формата с плавающей запятой)
precision(p);	Устанавливает точность p
w = width();	Возвращает текущее значение ширины поля (в символах)
width(w);	Устанавливает ширину текущего поля
setf(flags);	Устанавливает флаг форматирования (например, ios::left)
unsetf(flags);	Сбрасывает указанный флаг форматирования
setf(flags, field);	Очищает поле, затем устанавливает флаги форматирования

Примеры вызова функций в потоковом объекте `cout`:

- установка ширины поля 12 — `cout.width(12);`
- задание символа заполнения «звездочка» — `cout.fill('*');`
- установка выравнивания по левому краю — `cout.setf(ios::left);`
- восстановление выравнивания по правому краю — `cout.unsetf(ios::left);`

Порядок использования функции `setf(flags, field)`: вначале аргумент `field` задает сбрасываемые флаги, а затем аргумент `flags` указывает устанавливаемые флаги (табл. 18.10).

Таблица 18.10. Функция setf(flags, field)

Первый аргумент: устанавливаемые флаги	Второй аргумент: очищаемые флаги
`dec, oct, hex`	`basefield`
`left, right, internal`	`adjustfield`
`scientific, fixed`	`floatfield`

Например, вызов функции `cout.setf(ios::left, ios::adjustfield);` сбрасывает все флаги, связанные с выравниванием текста, а затем устанавливает флаг `left` для выравнивания по левому краю.

Описанные средства форматирования ввода-вывода применимы как к стандартным, так и к файловым потокам.

Ошибки потоков

Флаги статуса ошибок потоков определяют компонент `ios enum`, который сообщает об ошибках, произошедших в операциях ввода-вывода (табл. 18.11). Для чтения (и установки) флагов применяются специальные функции `ios` (табл. 18.12).

Таблица 18.11. Флаги статуса ошибок

Название	Значение
`goodbit`	Ошибок нет (флаги не установлены, значение = 0)
`eofbit`	Достигнут конец файла
`failbit`	Операция не выполнена (пользовательская ошибка, преждевременный EOF)
`badbit`	Недопустимая операция (нет ассоциированного `streambuf`)
`hardfail`	Неисправимая ошибка

Таблица 18.12. Функции для флагов ошибок

Функция	Назначение
`int = eof();`	Возвращает `true`, если установлен флаг EOF
`int = fail();`	Возвращает `true`, если установлены флаги `failbit`, `badbit` или `hardfail`
`int = bad();`	Возвращает `true`, если установлены флаги `badbit` или `hardfail`
`int = good();`	Возвращает `true`, если ошибки не было
`clear(int=0);`	При использовании без аргумента снимает все флаги ошибок, в противном случае устанавливает указанный флаг, например `clear(ios::failbit)`

Ввод чисел

Рассмотрим обработку ошибки при вводе чисел с клавиатуры или с диска. Самый общий источник ошибок — ввод символов вместо цифр. Здесь проверяется флаг `goodbit`. Если его значение не равно `true`, пользователю предлагается ввести правильное число.

```
while(true)    //Цикл до тех пор, пока ввод не будет правильным
    {
    cout <<"\nВведите целое число: ";
    cin >> i;
    if( cin.good() )    //если нет ошибок
        {
        cin.ignore(10, '\n');//удалить разделитель строк
        break;
        }           //выйти из цикла
    cin.clear();    //очистить биты ошибок
    cout << "Неправильный ввод данных";
    cin.ignore(10, '\n');        //удалить разделитель строк
    }
cout << "целое число: " << i;    //целое без ошибок
```

Избыток символов

Как правило, это происходит при передаче входного потока с ошибками. Лишние символы продолжают оставаться в потоке после завершения ввода. Затем они проникают в следующую операцию ввода. Часто остается не принятым символ новой строки. Чтобы избежать случайного извлечения из потока лишних символов, используется функция класса `istream` по имени `ignore(MAX, DELIM)`. Она считывает и удаляет из входного потока вплоть до `MAX` символов, включая ограничитель, указанный как аргумент `DELIM`. В вышеприведенной программе выражение `cin.ignore(10, '\n');` приводит к считыванию до 10 символов, включая ограничитель `'\n'`, и удалению их из входного потока.

Файловые потоки

При работе с файлами потоковые объекты могут открываться в нескольких режимах (табл. 18.13). Эти режимы задаются в классе `ios` и дублируются в производных от него классах. По сути, режим является атрибутом файла, а не потока.

Таблица 18.13. Биты режимов

Бит режима	Результат
`in`	Открытие для чтения (по умолчанию для `ifstream`)
`out`	Открытие для записи (по умолчанию для `ofstream`)
`ate`	Чтение, начиная с конца файла (AT End)
`app`	Запись, начиная с конца файла (APPend)
`trunc`	Обрезать файл до нулевой длины, если он уже существует (TRUNCate)
`nocreate`	Не открывать несуществующий файл
`noreplace`	Не открывать для вывода существующий файл, если не установлены `ate` или `app`
`binary`	Открыть в двоичном (не текстовом) режиме

Каждый раз, когда открывается файл, либо при обращении к функции `open()`, либо при инициализации потока именем файла, необходимо задать режим файла. Для каждого класса из заголовочного файла `fstream` определен набор возможных режимов. Подобно флагам состояния, режимы файла указываются целочисленными константами, которые можно комбинировать, применяя побитовые операции для установки нескольких режимов при открытии конкретного файла. Конструкторы файлового потока и функция `open()` имеют заданный по умолчанию аргумент, устанавливающий режим файла. Это значение по умолчанию варьируется в зависимости от типа потока. В качестве альтернативы режим файла можно указать явно.

Режимы `out`, `trunc` и `app` определены лишь для файлов, связанных с объектами класса `ofstream` или `fstream`, а режим `in` возможен только для файлов, подключенных к объектам класса `ifstream` или `fstream`. В режиме `ate` или `binary` можно открывать любой файл. Режим `ate` обеспечивает переход в конец файла только в момент открытия. В режиме `binary` файл рассматривается как последовательность байтов, символы в подключенном потоке при этом никак не интерпретируются.

По умолчанию файлы, связанные с объектом класса `ifstream`, открываются в режиме `in`, который позволяет считывать их данные.

Файлы, подключенные к объекту класса `ofstream`, открываются в режиме `out`, который разрешает записывать в них данные. Файл, открытый в режиме `out`, усекается, то есть все хранимые данные уничтожаются. Фактически указание режима `out` для объекта класса `ofstream` эквивалентно заданию пары режимов `out` и `trunc`.

Единственный способ сохранить существующие данные в файле, подключенном к объекту класса `ofstream`, состоит в явном указании режима `app`.

```
// режим вывода по умолчанию; усечение файла по имени "file_1"
ofstream of1("file_1");
// эквивалентное действие: файл "file_1" усечен явно
ofstream of2(''file_1", ofstream: :out | ofstream::trunc);
// режим app; новые данные добавляются в конец существующего
                    // файла по имени "file_2"
ofstream af("file_2", ofstream::app);
```

В определении потокового объекта `of2` записана побитовая операция `or`, чтобы открыть файл и в режиме `out`, и в режиме `trunc`.

Потоковый объект класса `fstream` может одновременно и читать, и записывать данные в подключенный файл. Способ воздействия такого объекта на свой файл зависит от режима, указанного при открытии файла.

По умолчанию объект класса `fstream` работает в режимах `in` и `out`. Его файл не усекается. Если файл, подключенный к объекту класса `fstream`, открыть только в режиме `out`, без режима `in`, файл будет усечен. Файл усекается в случае, если задан режим `trunc`, независимо от того, указан ли режим `in`. В приведенном примере файл `Example` открывается как для ввода, так и для вывода:

```
// открыть для ввода и вывода
fstream inOut("Example", fstream::in | fstream::out);
```

Режим устанавливается каждый раз, когда открывается файл:

```
ofstream of;
// установлен режим вывода out, файл "sample" усекается
of.open("sample", ofstream::out);
of.close(); // файл закрыт, его можно подключить снова
// открыть файл "file" в режиме app
of.open("file", ofstream::app);
of.close();
// режим вывода по умолчанию, файл "file_2" усекается
of.open("file_2");
```

При первом вызове функции `open()` указан режим `ofstream::out`. В результате файл по имени `sample` будет открыт в режиме вывода и усечен. Файл `file` открывается в режиме добавления. Все находящиеся в нем данные остаются, а новые записи добавляются в конец. При открытии файла `file_2` режим вывода не был задан явно. В итоге файл открывается в режиме `out`, а находившиеся в нем ранее данные уничтожаются.

При каждом вызове функции `open()` режим для файла задается либо явно, либо неявно. Если режим не установлен явно, используется значение по умолчанию.

Не все режимы могут задаваться одновременно. Некоторые из их комбинаций не имеют смысла, например `in` и `trunc`. При такой комбинации файл открывается для чтения, но находившиеся в нем данные предварительно уничтожаются. В результате читать просто нечего. Перечислим допустимые комбинации режимов:

- `out` — файл открыт для вывода. Все прежние данные удаляются;
- `out | app` — файл открыт для вывода. Все новые данные добавляются в конец;
- `out | trunc` — аналогично режиму `out`;
- `in` — файл открыт для ввода;
- `in | out` — файл открыт для ввода и вывода. Указатель чтения устанавливается в начало файла;
- `in | out | trunc` — файл открыт для ввода и вывода. Все прежние данные удаляются.

Любая из этих комбинаций может также содержать режим `ate`. Добавление режима `ate` задает переход к концу файла перед первой операцией ввода или вывода. Режим `ate` изменяет лишь начальную позицию файла.

Обсудим работу с числовыми файлами. Форматированный файловый ввод-вывод чисел приемлем лишь для малого количества небольших чисел. Причина такой рекомендации кроется в больших затратах памяти на хранение символьных строк. Например, текстовая версия целого числа «21947» занимает в памяти 5 байтов, а вещественного числа «5,17314e12» — 10 байтов. В то же время их двоичные представления расходуют по 4 байта памяти. Следовательно, механизм двоичного ввода-вывода существенно эффективнее.

В качестве первого примера рассмотрим программу двоичной записи и чтения из файла массива целых чисел. Запись будем производить с помощью функции `write()` из класса `ofstream`, а чтение — с помощью функции `read()` из класса `ifstream`.

Функции класса `ofstream` перечислены в табл. 18.14, а класса `ifstream` — в табл. 18.15.

Таблица 18.14. Функции класса ostream

Функция	Назначение
`<<`	Форматированная вставка данных любых стандартных (и перегруженных) типов
`put(ch)`	Вставка символа `ch` в поток
`flush ()`	Очистка буфера и вставка разделителя строк
`write (str, SIZE)`	Вставка `SIZE` символов из массива `str` в файл
`seekp(position)`	Установка позиции (в байтах файлового указателя) относительно начала файла
`seekp(position, seek_dir)`	Установка позиции (в байтах файлового указателя) относительно указанного места в файле. `seek_dir` может принимать значения `ios::beg, ios::cur, ios::end`
`pos = tellp()`	Возвращение позиции указателя файла в байтах

Таблица 18.15. Функции класса istream

Функция	Назначение
`>>`	Форматированное извлечение данных всех основных (и перегружаемых) типов из потока
`get(ch)`	Извлечение одного символа в `ch`
`get(str)`	Извлечение символов в массив `str` до ограничителя `'\n'`
`get(str, MAX)`	Извлечение в массив символов (их количество определяется числом `MAX`)
`get(str, DELIM)`	Извлечение символов в массив `str` до указанного ограничителя (обычно `'\n'`). Оставляет ограничитель в потоке
`get(str, MAX, DELIM)`	Извлечение в массив `str` до `MAX` символов или до символа `DELIM`. Оставляет ограничитель в потоке
`getline (str, MAX, DELIM)`	Извлечение в массив `str` до `MAX` символов или до символа `DELIM`. Извлекает ограничитель из потока
`putback (ch)`	Вставка последнего прочитанного символа обратно во входной поток
`ignore (MAX, DELIM)`	Извлечение и удаление до `MAX` символов, до ограничителя включительно (обычно `'\n'`). С извлеченными данными ничего не делает
`peek (ch)`	Чтение одного символа, символ остается в потоке
`count = gcount()`	Возвращение такого числа символов, которые прочитаны только что встретившимися вызовами `get()`, `getline()` или `read()`
`read (str, MAX)`	Для файлов. Извлечение вплоть до `MAX` символов в массив `str`; обработка выполняется до признака конца файла (EOF)
`seekg()`	Установка расстояния (в байтах) от начала файла до файлового указателя
`seekg(pos, seek_dir)`	Установка расстояния (в байтах) от указанной позиции в файле до указателя файла. `seek_dir` может принимать значения `ios::beg, ios::cur, ios::end`
`pos = tellg (pos)`	Возвращение позиции (в байтах) указателя файла от начала файла

Функции `write()` и `read()` обрабатывают данные в формате байтов (тип `char`): они просто переносят байты из буфера в файл и обратно. Параметрами этих функций являются адрес буфера и его длина. Поскольку исходный адрес буфера определен для массива чисел типа `int`, его адрес должен быть приведен к типу `char*`. Эта задача решается с помощью функции *непреобразующего явного приведения типа*, имеющей следующий формат:

```
reinterpret_cast <имя_целевого_типа> (имя_исходного_типа)
```

В нашем случае достаточно записать:

```
reinterpret_cast <char*> (buffer)
```

Вторым параметром функций записи и чтения является длина буфера в байтах, а не число элементов в буфере. Если в буфере хранится `MAX` элементов типа `int`, то длина вычисляется по выражению `MAX * sizeof(int)`.

Для вывода используем потоковый объект `os` класса `ofstream`, а для ввода — потоковый объект `is` класса `fstream`. В момент открытия потокового объекта свяжем его с физическим файлом `fdata.dat` и определим режим работы как `binary`.

Текст нашей программы имеет следующий вид:

```
// bin_file_io.cpp
// Двоичный файловый ввод-вывод целочисленных данных
#include <fstream>                //для файловых потоков
#include <iostream>
using namespace std;
const int MAX = 500;              //размер буфера
int buffer[MAX];                  //буфер для целых чисел

int main()
    {
    for(int j=0; j<MAX; j++)      //заполнить буфер числами
        buff[j] = j;              //(0, 1, 2, ...)
                                  //создать выходной поток
    ofstream os("fdata.dat", ios::binary);
                                  //записать в него
    os.write(reinterpret_cast<char*>(buffer), MAX*sizeof(int) );
    os.close();                   //должен закрыть его

    for(j=0; j<MAX; j++)          //стереть буфер
        buff[j] = 0;
                        //создать входной поток
    ifstream is("fdata.dat", ios::binary);
                                  //читать из него
    is.read( reinterpret_cast<char*>(buffer), MAX*sizeof(int) );

    for(j=0; j<MAX; j++)          //проверка данных
        if( buff[j] != j )
            { cerr << "Неправильные данные!\n"; return 1; }
    cout << "Данные правильны\n";
    return 0;
    }
```

Заметим, что в программе два потока (входной и выходной) связаны с одним и тем же файлом `fdata.dat`, поэтому первый поток должен быть закрыт до того, как откроется второй. Для этого используется функция `close()` потокового объекта `os`,

который работает первым. Когда программа завершается, второй объект `is` вызывает свой деструктор, закрывающий файл автоматически, без указаний программиста.

В качестве второго примера обсудим программу, обеспечивающую запись в файл объектов, а также чтение объектов из файла. Применим один универсальный потоковый объект `oism`, способный поддерживать как вывод, так и ввод. Для этого он должен быть экземпляром класса `fstream`, являющегося наследником класса `iostream`. Кроме того, отделим задачу создания потокового объекта от задачи его открытия. Открывать поток будем с помощью функции `open()` из класса `fstream`. Тем самым мы улучшаем диагностику ошибки при открытии. Появляется возможность один раз создать объект, а затем многократно пытаться открыть его, не заботясь каждый раз о создании нового. Режим работы файла, присоединяемого к объекту, определим с помощью нескольких флагов:

- Флаг `ios::app` позволяет при новой записи сохранять все, что было записано в файл прежде.
- Флаги `ios::in` и `ios::out` указывают на возможность и ввода, и вывода.
- Флаг `ios::binary` свидетельствуют о работе с двоичными объектами.

Все эти флаги применяются одновременно.

Теперь представим текст программы:

```
// objects_io.cpp
// Чтение из файла и запись нескольких объектов
#include <fstream>              //для файловых потоков
#include <iostream>
using namespace std;
//---
class student                   //класс студент
    {
    private:
        char name[80];          //имя студента
        float average_mark;     //средний балл
    public:
        void get()              //получить данные о студенте
            {
            cout << "\n    Введите имя: "; cin >> name;
            cout << "    Введите средний балл: "; cin >> average_mark;
            }
        void put()              //вывод данных о студенте
            {
            cout << "\n    Имя: " << name;
            cout << "\n    Средний балл: " << average_mark;
            }
    };
//---
int main()
    {
    char ch;
    student stnt;               //создать объект stnt
    fstream oism;               //создать входной/выходной поток
//открыть для добавления
    oism.open("group.dat", ios::app | ios::out |
                                          ios::in | ios::binary );
    do                          //данные от студента – в файл
```

```
    {
    cout << "\nВведите данные о человеке:";
    stnt.get();                 //получить данные
                                //записать их в файл
    oism.write( reinterpret_cast<char*>(&stnt), sizeof(stnt) );
    cout << "Продолжить ввод (y/n)? ";
    cin >> ch;
    }
while(ch=='y');                 //выход по 'n'
oism.seekg(0);          //поставить указатель на начало файла
                        //считать данные о первом человеке
oism.read( reinterpret_cast<char*>(&stnt), sizeof(stnt) );
while( !oism.eof() )            //Выход по EOF
    {
    cout << "\nСтудент:";       //вывести данные
    stnt.put();                 //считать данные о следующем
    oism.read(reinterpret_cast<char*>(&stnt),sizeof(stnt));
    }
cout << endl;
return 0;
}
```

Предполагается, что программа предварительно запускалась и были введены и записаны в файл данные, например, о семи студентах.

За один проход программы в файл с помощью функции `write()` записывается один объект класса `student`. После окончания записи появляется необходимость прочитать весь файл `group.dat`. Для этого с помощью функции `seekg()` указатель файла устанавливается на начало. После чего чтение данных начинается с начала файла. В цикле `while` из файла считывается очередной объект, и его данные выводятся на экран. Цикл продолжается до тех пор, пока не будут прочитаны все объекты класса `student`. Момент завершения индицируется флагом `eofbit`.

Строковые потоки

Достаточно часто удобно считать потоком область памяти. В этом случае в нее можно записывать данные точно так же, как и в файл. В программах такие области памяти определяются как символьные массивы — строки.

Семейство потоковых классов поддерживает работу с такими строками в памяти. Для вывода в память строки существует специальный класс `ostringstream`, наследник класса `ostream`. Для ввода из памяти строки служит класс `istringstream`, порожденный классом `istream`. Для объектов, которым требуется выполнять одновременно ввод и вывод, сформирован класс `stringstream`, являющийся наследником `iostream`.

Чтобы воспользоваться любым из этих классов, необходимо подключить к программе заголовочный файл `sstream.h`.

Кроме унаследованных возможностей, строковые классы имеют конструктор, аргументом которого является объект класса `string`, и переменную-член по имени `str`, обеспечивающую доступ к обрабатываемой строке. Строковый поток ввода полезен для извлечения числовых значений из строк. Напротив, строковый поток вывода удобен для форматирования вывода в GUI-средах, функции которых ожи-

дают аргумента в виде простой строки. Как правило, строковый поток ввода инициируется объектом класса `string`, а затем считывает из него символы, используя операции ввода. И наоборот, строковый поток вывода инициируется пустым объектом класса `string`, а затем заполняется с помощью операций вывода. Строковый поток ввода-вывода обычно используется для отделения собственно ввода-вывода от обработки данных. Например, аргумент типа `string` в функции `str_to_double()` обычно поступает из файла (журнала веб-событий или с клавиатуры). Аналогично сообщение, составленное функцией `my_code()`, в конце концов выводится на экран. Таким образом, строковые потоки можно интерпретировать как механизм настройки ввода-вывода для особых потребностей и вкусов.

Перечислим операции, специфические для класса `stringstream`:

- `stringstream strm;` — создает несвязанный объект класса `stringstream`;
- `stringstream strm(s);` — создает объект класса `stringstream`, содержащий копию строки `s`;
- `strm.str()` — возвращает копию строки, которую хранит объект `strm`;
- `strm.str(s)` — копирует строку `s` в объект `strm`.

Строковый класс обеспечивает обработку как по целым строкам, так и по отдельным словам внутри каждой строки:

```
string line, word;              // объекты для хранения введенных строк и слов
while (getline(cin, line)) {    // читать введенные строки в переменную line
// осуществить построчную обработку
       istringstream pipeline(line);     // связать поток с читаемой строкой
       while (pipeline >> word) {        // читать слова из строки
                                 // осуществить обработку по одному слову
    }
}
```

В данном случае целая строка вводится с помощью функции `getline()`. Для получения доступа к каждому ее слову объект `pipeline` класса `istringstream` подключается к прочитанной строке. После этого для чтения слов из каждой строки можно использовать обычную операцию ввода.

Самым популярным применением строковых потоков является автоматическое преобразование типов данных. Например, может возникнуть необходимость представить коллекцию числовых значений в виде строк или наоборот. Операции ввода и вывода строковых классов автоматически преобразуют числовое значение в соответствующее ему строковое представление, и наоборот:

```
int a = 256, b = 4096;
ostringstream formatter;
// значения преобразуются в строковое представление
formatter << "a: " << a << " \n "
            << "b: " << b << " \n ";
```

В этом фрагменте создается пустой объект класса `ostringstream` по имени `formatter`, и в него помещается необходимый текст. Главное в том, что значения типа `int` автоматически преобразуются в их строковые эквиваленты. Следовательно, объект `formatter` содержит следующие символы:

```
a: 256\nb: 4096
```

При помощи потокового объекта класса `istringstream` можно получать числовое значение из прочитанной строки. Символьное представление, прочитанное объектом класса `istringstream`, автоматически преобразуется в соответствующее числовое значение.

```
// переменная-член str получает строку, связанную
// с объектом класса ostringstream
istringstream input_str(formatter.str());
string pool;     // поместить в строку pool отформатированное
                 // сообщение
// извлечь хранимые символьные значения и преобразовать их
// обратно, в числовые значения
input_str >> pool >> a >> pool >> b;
cout << a << " " << b << endl; // отобразит 256 4096
```

Переменная-член `str` обеспечивает доступ к копии строки, связанной с ранее созданным объектом класса `ostringstream`. В свою очередь, к этой строке подключается объект `input_str`. При чтении значений объекта `input_str` они преобразуются обратно в их первоначальные числовые представления.

Для чтения содержимого объекта `input str` необходимо анализировать элементы его строки. Для получения числовых значений надо выявить (и проигнорировать) метки, применяемые для разделения элементов данных.

Операция ввода читает типизированные значения, поэтому типы объектов-приемников должны быть совместимы с типами значений, читаемых из потокового объекта. В этом примере объект `input_str` содержит четыре элемента: строковое значение `a:`, далее `256`, затем строка `b:` и, наконец, `4096`. Как правило, при чтении строки с использованием операции ввода пробелы игнорируются. Следовательно, при чтении строки, подключенной к объекту `formatter`, можно игнорировать имеющиеся символы перехода на новую строку.

Контрольные вопросы и упражнения

1. Что понимается под файлом?
2. Назовите характерные особенности файлового типа данных.
3. Как объявить в программе файловую переменную?
4. Каким образом можно указать в программе местонахождение файла?
5. Назовите стандартные процедуры для подготовки (открытия) файла к чтению и к записи информации.
6. Какие операции могут быть выполнены при работе с файлами последовательного доступа?
7. Поясните порядок создания и обработки файлов последовательного доступа.
8. Поясните порядок создания и обработки текстового файла.
9. Поясните порядок создания и обработки файлов прямого доступа.
10. Сравните файл последовательного доступа с текстовым файлом

Глава 19

Основные понятия параллельного программирования

Эта глава начинается традиционно: с описания основных понятий параллельного программирования — процессов, потоков и их взаимодействия. После чего в терминах языка Ада рассматриваются следующие классические проблемы: синхронизация процессов с помощью разделяемых переменных (на основе семафоров, мониторов, защищенных объектов); синхронизация процессов с помощью сообщений; «развязка» взаимодействия процессов при рандеву. Затем обсуждаются потоки в языках Java и C#.

Параллельные действия — реальная часть наиболее интересных программ. Например, операционная система с разделением времени должна иметь дело с рядом пользователей, работающих одновременно на их терминалах. Далее, большинство приложений реального времени, которые управляют физическими процессами, составлены из параллельных программных сегментов, причем каждый сегмент отвечает за свою собственную физическую подсистему. Наконец, мир параллелен, заполнен людьми, делающими различные вещи в одно и то же время, и программная модель мира должна составляться из параллельных сегментов.

Параллельное программирование — это нотация программирования и средства для выражения потенциального параллелизма и решения проблем синхронизации и коммуникации (взаимодействия) [3, 12, 13, 26, 31, 32, 38, 39, 44, 45, 54, 95].

Реализация параллелизма — отдельная тема (в компьютерных системах), в значительной степени независимая от параллельного программирования. Параллельное программирование обеспечивает формирование абстракций для изучения параллелизма, без отображения деталей реализации.

Процессы и потоки

Процесс (от лат. processus — продвижение) — это последовательная смена состояний объекта во времени. Природа объекта может быть произвольной: материальный (природный или искусственный) или идеальный (понятие, теория и т. д.) объект порождает соответственно материальный или идеальный процесс (например, процесс приготовления пищи, процесс любовных переживаний).

Международный стандарт «ISO 9000: 2005 Системы менеджмента качества. Основные положения и словарь» определяет *процесс* как совокупность взаимо-

связанных или взаимодействующих видов деятельности, преобразующую входы в выходы.

Различают дискретные и непрерывные процессы. У непрерывных процессов между двумя любыми состояниями всегда можно выделить промежуточные; таким образом, понятие соседних состояний к подобному процессу неприменимо (можно говорить лишь о последовательных состояниях и о близости их друг другу во времени). Примерами непрерывных процессов являются кипение воды, движение планет вокруг Солнца и т. д.

У дискретных процессов состояния четко отграничены друг от друга, и для каждого состояния можно указать соседние (предыдущее и последующее или только одно из них). Примером дискретного процесса является работа кассира в магазине (можно выделить состояния получения денег, выдачи сдачи, выдачи чека и т. д.).

Рассматривают также апериодические и циклические процессы (в последних одни и те же состояния повторяются во времени в одной и той же последовательности), конечные (с известным или неизвестным временем протекания) и бесконечные (в некотором абстрактном смысле), управляемые и неуправляемые, линейные (однонаправленные) и разветвленные процессы. Эти и другие типы процессов выделяют в зависимости от задач, которые решаются при изучении или использовании процессов.

Программный процесс является абстрактным понятием, описывающим работу программы. Процесс образует выполняемая программа, задействующая некоторые ресурсы, включая текущие значения счетчика команд, регистров и переменных. С позиций данной модели, у каждого процесса есть собственный виртуальный центральный процессор. Различие между процессом и программой имеет принципиальное значение. Можно сказать, что программа — это подробная инструкция на тему «как решить задачу», а процесс — это последовательность действий виртуального компьютера для ее решения, использующая данную инструкцию.

Итак, процесс — это определенного рода активность. У него есть программа, входные и выходные данные, а также состояние. Один аппаратный процессор может переключаться между различными процессами.

Состояния программного процесса иллюстрирует диаграмма на рис. 19.1, где вершины обозначают состояния, а стрелки — переходы между состояниями.

В состоянии «Создается» процесс получает от системы ресурсы, обеспечивающие его деятельность, в состоянии «Инициализируется» формируются начальные значения рабочих объектов процесса, состояние «Исполняется» соответствует вычислениям, которые производит процесс, а в состоянии «Прекращается» процесс готовится к переходу (возврату) в пассивную фазу (финализируются объекты процесса, освобождаются выделенные ресурсы).

Параллельная программа задает набор автономных последовательных процессов, выполняемых (логически) параллельно. Плоская модель параллельных процессов, время от времени взаимодействующих друг с другом, соответствует простейшему случаю.

В общем случае создается иерархия процессов со сложной системой взаимодействия.

Обычно для каждого процесса различают процесс (блок), который создает его, и процесс (блок), на который влияет его завершение.

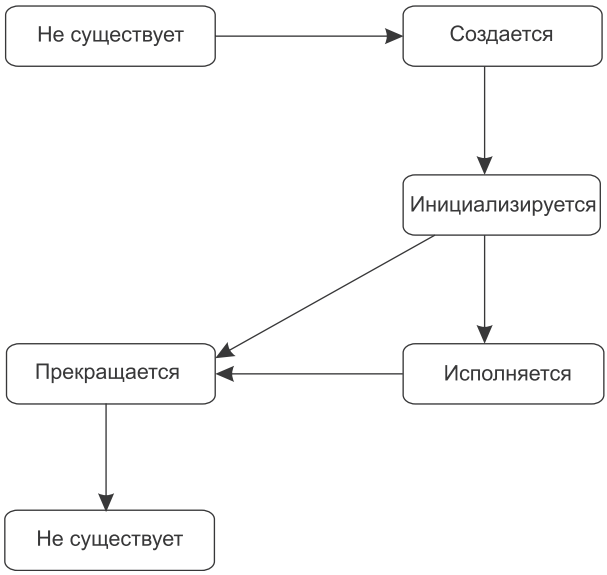


Рис. 19.1. Состояния последовательного программного процесса

Первое отношение известно как «*родитель-потомок*», родитель задерживается на время создания и инициализации потомка.

Второе отношение называют «*мастер-слуга*». Процесс может зависеть или от всего процесса-мастера, или от его внутреннего блока.

Процессу-мастеру не разрешают выходить из блока, пока не будут завершены все зависимые от этого блока процессы.

В частном случае процесс-родитель может одновременно быть и мастер-процессом для потомка.

Как показано на рис. 19.2, с учетом иерархических отношений количество состояний процесса увеличивается.

Процесс удобно рассматривать как способ объединения родственных ресурсов в одну группу. У каждого процесса есть свое адресное пространство в памяти, содержащее саму программу и данные, а также другие ресурсы. Ресурсами являются открытые файлы, дочерние процессы, необработанные аварийные сообщения, обработчики сигналов, учетная информация и многое другое. Гораздо проще управлять ресурсами, объединив их в форме процесса.

Кроме того, процесс является потоком исполняемых команд или просто потоком. Говорят, что у процесса имеется свой собственный, одиночный поток управления. У потока есть счетчик команд, отслеживающий порядок выполнения действий. У него есть регистры, в которых хранятся текущие переменные. У него есть стек, содержащий активации подпрограмм процесса, где на каждую подпрограмму, вызванную, но еще не завершенную, отведена отдельная запись. Хотя поток должен исполняться внутри процесса, целесообразно различать понятия потока и процесса. Процессы используются для группирования ресурсов, а потоки являются последовательностями объектов, поочередно исполняющихся на центральном процессоре.

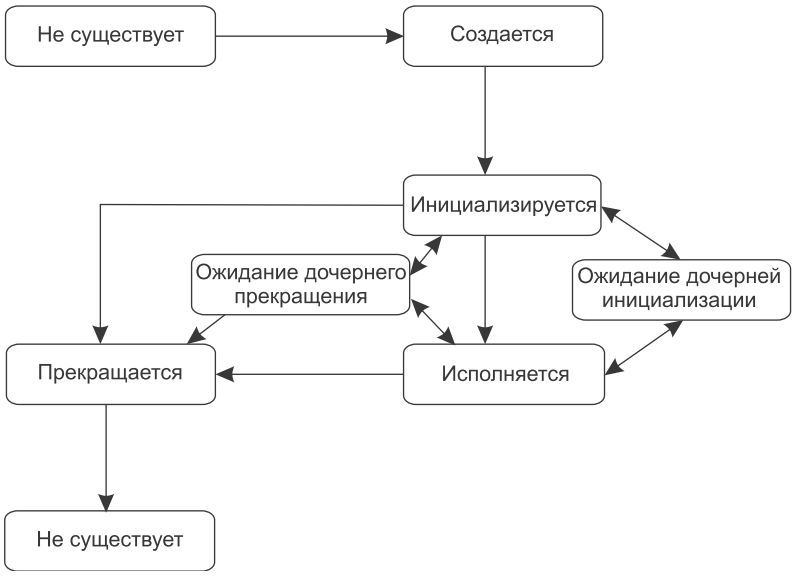


Рис. 19.2. Состояния программного процесса при учете иерархии отношений

Модель потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких независимых фрагментов программы. Несколько потоков, работающих параллельно в одном процессе, подобны нескольким процессам, идущим параллельно на одном компьютере. В первом случае потоки разделяют адресное пространство, открытые файлы и другие ресурсы. Во втором случае процессы совместно пользуются физической памятью, дисками, принтерами и другими ресурсами. Потоки обладают некоторыми свойствами процессов, поэтому их иногда называют упрощенными, облегченными процессами. Важно помнить, что у каждого потока существует свой собственный стек выполнения. Идею использования нескольких потоков в одном процессе представляют термином «*многопоточность*». При запуске многопоточного процесса в однопроцессорном компьютере потоки работают поочередно.

Как показано в табл. 19.1, потоки разделяют не только адресное пространство, но и открытые файлы, дочерние процессы, сигналы и многое другое.

Таблица 19.1. Ресурсы процесса и его потоков

Ресурсы процесса	Ресурсы потока
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные исключения	
Сигналы и их обработчики	
Информация об использовании ресурсов	

Ресурсы процесса находятся в совместном использовании всеми потоками процесса; кроме того, каждый поток имеет индивидуальные ресурсы. Например, если один поток открывает файл, этот файл тут же становится доступным для остальных потоков, и они могут считывать информацию и записывать ее в файл. Это естественно, поскольку процесс, а не поток является единицей управления ресурсами. Если бы каждый поток обладал собственным адресным пространством, открытыми файлами и т. д., он назывался бы отдельным процессом. Идея многопоточности состоит в совместном использовании ресурсов несколькими потоками для выполнения *общей задачи* и в *тесном взаимодействии*.

По сравнению с различными процессами степень зависимости у различных потоков в одном процессе увеличена. У всех потоков одно и то же адресное пространство, они совместно используют глобальные переменные. Поскольку любой поток имеет доступ к любой ячейке памяти в адресном пространстве процесса, один поток может считывать, записывать или даже затирать информацию из стека другого потока.

Задачи языка Ада

В языке Ада единицу распараллеливания называют *задачей* (task). Задачи должны объявляться явно. Задачи могут объявляться на любом программном уровне; создаются они неявно, при входе в область действия их объявления.

Задача может быть объявлена как тип или как единичный экземпляр (анонимного типа).

Задача-тип состоит из спецификации и тела.

Спецификация содержит:

- имя типа;
- необязательную дискриминантную часть, определяющую параметры, которые могут поступать в экземпляры задачного типа (во время их создания);
- видимую часть, которая определяет входы и предложения представления;
- приватную часть, которая определяет скрытные входы и предложения представления.

Вход именует услугу, которую задача предлагает клиентам (другим задачам). Тело задачи содержит последовательность исполняемых операторов.

Примеры спецификаций задач приведены в табл. 19.2.

Таблица 19.2. Спецификации задач

Спецификация	Пояснение

`task type Server (Init : Parameter) is`
`  entry Service;`
`end Server;`
	Этот задачный тип имеет дискриминантную часть с параметром Init и вход с именем Service

`task type Controller;`	Этот задачный тип не имеет входов; он не может прямо взаимодействовать с другими задачами
`task type Agent(Param : Integer);`	Этот задачный тип не имеет входов, но объекты-задачи могут получать во время создания целый параметр

Спецификация	Пояснение
`task type Гаражное_Обслуживание (Заправка:Число_заправок := 1) is entry Заправка_бака(G: Gallons); entry Частич_Заправка(G: Gallons); end Гаражное_Обслуживание;`	Экземпляры этого типа будут взаимодействовать через два входа; сохраняемое количество заправок поступает во время создания задачи; если значение не задается, используется значение по умолчанию — 1

Примеры объявления задач:

```
Main_Controller : Controller;
Обслуживание1: Гаражное_Обслуживание(2);-- обслуж. 2 запр.
Input_Analyser : Character_Count(30, Nick);
type Сеть_Гаражей is array (1 .. 10) of Гаражное_Обслуживание;
GN : Сеть_Гаражей;
```

Выполнение объекта-задачи включает три этапа:

- *Активация* — обработка объявлений декларативной части, тела задачи (на этом этапе создаются и инициализируются все локальные переменные задачи).
- *Обычное выполнение* — выполнение операторов из тела задачи.
- *Финализация* — выполнение кода финализации, ассоциированного с объектами в ее декларативной части.

Все статические задачи описываются в единой секции объявления программы, их активация начинается непосредственно при обработке секции.

Первый оператор, следующий за секцией объявления, не выполняется до тех пор, пока не завершится активация всех задач.

После активации выполнение задачи-объекта определяется содержанием его тела. Перед выполнением своего тела задача не ждет активации других задач-объектов.

Задача может попытаться взаимодействовать с другой задачей сразу после своего создания; вызывающая задача задерживается до тех пор, пока не будет готова вызываемая задача.

Пример 1. Процедура с двумя задачами.

```
procedure Example1 is
    task A;
    task B;
    task body A is
    -- локальные объявления задачи A
        begin
        -- последовательность операторов задачи A
        end A;
    task body B is
    -- локальные объявления задачи B
        begin
        -- последовательность операторов задачи B
        end B;
begin
    -- задачи A и B начинают работу перед
    -- первым оператором из раздела
    -- операторов процедуры.
    ...
end Example1; -- процедура не прекращается, пока  не прекратятся задачи A и B.
```

ПРИМЕЧАНИЕ

При обработке декларативной части процедуры активируются две задачи. При пересечении открывающей скобки begin раздела операторов стартуют три процесса (две задачи и сама процедура).

Динамические задачи создаются применением генератора new к ссылочному типу, обслуживающему задачный тип. Они активируются непосредственно после обработки генератором new. Задача, создаваемая генератором, блокируется до тех пор, пока все созданные задачи не закончат свою активацию.

Пример 2. Составной блок со статическими и динамическими задачами.

```
declare
task type T;
type A is access T;-- ссылочный тип на задачный тип
P : A;
Q : A:= new T; -- попытка активации и выполнения динамической задачи, вводится
                                                                задержка запуска
B, C : T; -- B и C создаются при  обработке объявления
task body T is  ...;
-- задачи активируются, когда обработка объявлений завершена
begin
    -- первый  оператор выполняется после завершения активации всех задач
    …
    P := new T; -- активизация и запуск динамической задачи
end;
```

ПРИМЕЧАНИЕ

При пересечении открывающей скобки begin операторов блока стартуют статические задачи B, C, сам блок и динамическая задача Q.all. Далее к ним присоединяется динамическая задача P.all.

Если при обработке декларативной части генерируется исключение, все задачи, создаваемые при ее обработке, никогда не активируются, а станут прекращенными.

Если исключение генерируется при активации задачи, задача становится завершенной или прекращенной, и перед первым исполняемым оператором декларативного блока генерируется предопределенное исключение Tasking_Error (исключение генерируется только один раз). Обработка этого исключения откладывается до тех пор, пока все активированные задачи не завершат их активацию.

Пример 3. Параллельные вычисления в блоке (и вычисления вообще) отменяются. Обрабатывается только сигнал исключения.

```
declare
A : TaskType1; -- успешно проходит свою активацию
B : TaskType2; -- генерирует исключение при активации
-- здесь задачи должны завершить активацию
begin
-- исполнение операторов должно начинаться после завершения активации задач
…-- вычисления отменяются
```

```
exception – срабатывает ловушка исключения
    when Tasking_Error =>…;
    when others =>…;
end;
```

Если между задачами существуют иерархические отношения, то в ходе параллельных вычислений имеют место задержки:

- Задача-родитель ожидает, когда завершится «рождение» (активация) процесса-ребенка.
- Задача-мастер ждет, когда процесс-слуга окончательно «прикажет долго жить» (финализируется).

Преждевременная кончина родителя-мастера приводит к уничтожению всех зависимых задач. Вот такие бытуют порядки в среде параллельных вычислений.

ПРИМЕЧАНИЕ

Мастером динамической задачи считают секцию, где был объявлен ссылочный тип, на основе которого эта задача создается.

Синхронизация процессов на основе разделяемых переменных

Корректность реализации параллельных вычислений зависит от синхронизации и взаимодействия между процессами-участниками.

Синхронизация — это выполнение ограничений путем чередования действий процессов (действие одного процесса выполняется только после действия другого).

Взаимодействие — передача информации от одного процесса к другому.

Понятия связаны: понятие взаимодействия требует синхронизации, а синхронизация может рассматриваться как взаимодействие без обмена данными.

Взаимодействие по данным обычно основывается или на разделяемых переменных, или на передаче сообщений.

Если процессы взаимодействуют через разделяемую переменную, они могут помешать друг другу. Для устранения помех работа одного процесса с разделяемой переменной не должна прерываться другим процессом. Иными словами, процессы должны работать с разделяемой переменной по очереди: один процесс должен ждать окончания работы другого процесса.

Последовательность операторов процесса, которые должны выполняться неразделимо, называется *критической секцией*.

Синхронизацию, требуемую для защиты критической секции, называют *взаимным исключением*.

Очень часто одного взаимного исключения бывает недостаточно. Приходится дополнительно вводить *условную синхронизацию*.

В качестве примера рассмотрим работу процессов с ограниченным буфером.

Ограниченный буфер имеет два очевидных условия синхронизации:

- Процессы-производители не должны пытаться класть данные в буфер, если буфер полон.
- Процессы-потребители не могут извлекать объекты из буфера, если буфер пуст.

Отсюда вывод: при обращении к буферу процесс должен проверить истинность «своего» условия синхронизации. Если условие не выполняется, процесс должен «встать в очередь» к буферу.

Классическими средствами обеспечения синхронизации являются семафоры и мониторы.

Семафоры

Основное назначение семафора отражает его название: синхронизировать прохождение процессами критических секций.

ПРИМЕЧАНИЕ

Очень похоже на синхронизацию движения поездов: семафор указывает на занятость железнодорожного перегона. Поезд допускается на перегон, если путь свободен (семафор открыт). Если семафор закрыт, поезд ждет своей очереди.

Семафор — неотрицательная целая переменная `S`, к которой применимы только две операции `Wait(S)` и `Signal(S)`:

- `Wait(S)`. Если значение `S > 0`, `тогда S := S - 1`; в противном случае процесс, вызывающий `Wait`, задерживается до получения `S > 0` (а затем уменьшает значение семафора).
- `Signal(S)`. Выполняет `S := S + 1`.

`Wait` и `Signal` атомарны (неделимы). Два процесса, выполняющие операции `Wait` над одним и тем же семафором, не могут помешать друг другу и не могут отказать в ходе выполнения семафорной операции.

Пример 1. Условная синхронизация.

```
var consyn : semaphore = 0;
process P1; -- ожидающий процесс
    операторX();
    wait (consyn); -- процесс стоит и ждет сигнала от процесса P2
    операторY();
end P1;
process P2; --сигнализирующий процесс
    оператор A();
    signal (consyn);
    операторB();
end  P2;
```

Пример 2. Взаимное исключение.

```
var mutex : semaphore = 1;
process P1;
    операторX();
    wait (mutex); -- процесс занимает путь
```

```
    операторY();
    signal (mutex); -- процесс освобождает путь
    оператор Z()
end P1;
process P2;
    операторA();
    wait (mutex); -- процесс занимает путь
    операторB();
    signal (mutex); -- процесс освобождает путь
    операторC();
end P2;
```

Пример 3. Организация доступа процессов к ограниченному буферу.

```
package Buffer is --разделяемый ресурс
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
end Buffer;
package body Buffer is
    Size : constant Natural := 32;
    type Buffer_Range is mod Size;
    Buf : array (Buffer_Range) of Integer;
    Top, Base : Buffer_Range := 0;
    Mutex : Semaphore(1);
    Item_Available : Semaphore(0);
    Space_Available : Semaphore(Size);
    procedure Append(I : Integer) is -- операция добавления элемента в буфер
    begin
        Wait(Space_Available); -- проверка условия синхронизации
        Wait(Mutex); -- занятие буфера
        Buf(Top) := I;
        Top := Top+1
        Signal(Mutex); -- освобождение буфера
        Signal(Item_Available); -- установка условия синхронизации
    end Append;
    procedure Take(I : out Integer) is -- операция изъятия элемента из буфера
    begin
        Wait(Item_Available); -- проверка условия синхронизации
        Wait(Mutex); -- занятие буфера
        I := BUF(base);
        Base := Base+1;
        Signal(Mutex); -- освобождение буфера
        Signal(Space_Available); -- установка условия синхронизации
    end Take;
end Buffer;
```

Выводы по семафорам

- Семафор — элегантный примитив низкоуровневой синхронизации, однако его использование может приводить к решениям, подверженным ошибкам.
- Если семафорная операция пропущена или неправильно размещена, разрушается вся программа. Когда программа имеет дело с редким, но критическим событием, не может гарантироваться взаимное исключение и возможна взаимная блокировка.
- Требуется более структурированный примитив синхронизации.

- Не существует высокоуровневых языков параллельного программирования, полностью зависящих от семафоров; они важны исторически, но не адекватны требованиям параллельных вычислений.

Мониторы

Мониторы обеспечивают инкапсуляцию критических секций и разделяемых переменных.

Критические секции записываются как операции, помещаемые в капсулу монитора. Для всех вызовов операций монитора гарантируется взаимное исключение. Все переменные, которые должны быть доступны в режиме взаимного исключения, скрыты. Извне видимы только операции монитора.

Правда, условную синхронизацию по-прежнему приходится организовывать вручную. Обычно для этого используют условные переменные.

Условная переменная `CV` обслуживается двумя семафороподобными операциями `Wait(CV)` и `Signal(CV)`:

- Процесс, вызывающий `Wait(CV)`, блокируется (приостанавливается) и помещается в очередь задержки, связанную с условной переменной `CV`. Блокированный процесс освобождает монитор, позволяя войти другому процессу.
- `Signal(CV)` освобождает один блокированный процесс (первый из очереди задержки). Если нет блокированных процессов, `Signal(CV)` не оказывает никакого действия.

Пример 4. Организация ограниченного буфера в виде монитора.

```
monitor buffer;
    var buf : array (1..N) of integer;
    top, base : 0..size-1;  NumberInBuffer : integer;
    spaceavailable, itemavailable : condition; -- условные переменные
    procedure append (I : integer);
    begin
        if NumberInBuffer = size then
            wait(spaceavailable); -- приостановка вызывающего процесса
        end if;
        buf(top) := I;
        NumberInBuffer := NumberInBuffer+1;
        top := (top+1) mod size;
        signal(itemavailable) -- освобождение первого из очереди
    end append;
    procedure take (var I : integer);
    begin
        if NumberInBuffer = 0 then
            wait(itemavailable); -- приостановка вызывающего процесса
        end if;
        I := buf(base);
        base := (base+1) mod size;
        NumberInBuffer := NumberInBuffer-1;
        signal(spaceavailable); -- освобождение первого из очереди
    end take;
begin -- инициализация
    NumberInBuffer := 0;
    top := 0; base := 0
end monitor;
```

ПРИМЕЧАНИЕ

Пример вымышленный, поскольку в языке Ада конструкции monitor нет.

Выводы по мониторам

- Монитор предлагает структурное и элегантное решение проблем взаимного исключения, возникающих, например, в ограниченном буфере.
- Тем не менее это не относится к условной синхронизации — здесь требуются низкоуровневые условные переменные.
- Все недостатки использования семафоров прямо распространяются на условные переменные.

Защищенные объекты

Защищенные объекты, средства языка Ада, являются усовершенствованной версией мониторов. Они сочетают преимущества мониторов с преимуществом условных критических секций:

- Инкапсулируют элементы данных и обеспечивают доступ к ним только с помощью защищенных действий — защищенных подпрограмм или защищенных входов.
- Операции имеют атомарное взаимное исключение.
- Для условной синхронизации используются сторожевые условия операций.

Защищенный объект может объявляться как тип или как единичный экземпляр.

Пример 5. Организация разделяемых данных в виде экземпляра — защищенного объекта.

```
protected Shared_Data_Item (Initial : Data_Item) is
                                -- спецификация защищенного объекта
    function Read return Data_Item;
    procedure Write (New_Value : in Data_Item);
private
    The_Data : Data_Item := Initial;
end Shared_Data_Item;
protected body Shared_Data_Item is -- тело защищенного объекта
    function Read return Data_Item is
    begin
        return The_Data;
    end Read;
    procedure Write (New_Value : in Data_Item) is
    begin
        The_Data := New_Value;
    end Write;
end Shared_Data_Item;
```

ПРИМЕЧАНИЕ

- Защищенная процедура Write обеспечивает доступ чтения/записи инкапсулированных данных в режиме взаимного исключения. Параллельные вызовы Write выполняются по одному в единицу времени.
- Защищенная функция Read обеспечивает только параллельный доступ чтения инкапсулированных данных. Параллельные вызовы Read могут выполняться одновременно.
- Вызовы процедур и функций защищенного объекта взаимно исключаются.
- Объявления типа данных в защищенном объекте запрещены.

Кроме процедур и функций, защищенный объект может содержать третью разновидность операций — *защищенный вход*.

Защищенный вход похож на защищенную процедуру в том, что его вызовы выполняются в режиме взаимного исключения и имеется доступ к данным по чтению/записи.

Защищенный вход охраняется булевым выражением, которое называется *барьером*. Если при вызове входа вычисляемый барьер равен `false`, вызывающая задача приостанавливается и остается в приостановленном состоянии, пока барьер имеет ложное значение. Следовательно, вызовы защищенного входа могут использоваться для реализации условной синхронизации.

Пример 6. Организация ограниченного буфера в виде защищенного типа.

```
Buffer_Size : constant Integer :=10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;
protected type Bounded_Buffer is -- спецификация
    entry Get (Item : out Data_Item);
    entry Put (Item : in Data_Item);
private
    First : Index := Index'First;
    Last : Index := Index'Last;
    Num : Count := 0;
    Buf : Buffer;
end Bounded_Buffer;
protected body Bounded_Buffer is -- тело
    entry Get (Item : out Data_Item) when
        Num /= 0 is -- это барьер
    begin
         Item := Buf(First);
        First := First + 1;
        Num := Num - 1;
    end Get;
    entry Put (Item : in Data_Item) when
        Num /= Buffer_Size is -- это барьер
    begin
        Last := Last + 1;
        Buf(Last) := Item
        Num := Num + 1;
    end Put;
end Bounded_Buffer;
My_Buffer : Bounded_Buffer; -- объявление защищенного объекта
```

Для вызова защищенного объекта указывается его имя и имя подпрограммы или входа:

```
My_Buffer.Put(Some_Data);
```

В любой момент времени барьер входа защищенного объекта или открыт, или закрыт; он открыт, если значение булевого выражения равно `true`, в противоположном случае он закрыт.

Барьеры вычисляются в следующих ситуациях:

- задача вызывает один из защищенных входов, а связанный барьер ссылается на переменную или атрибут, который должен измениться с момента последнего вычисления барьера;
- задача выполняет и покидает защищенную процедуру или вход, и имеются задачи, стоящие в очереди к входам, чьи барьеры ссылаются на переменные или атрибуты, которые должны измениться с момента последнего вычисления барьеров.

После вызова функции защищенного объекта повторно вычислять барьер не имеет смысла.

Очень интересную возможность предоставляет атрибут `'Count` защищенного входа. Атрибут `'Count` определяет количество задач в очереди ко входу. Его вычисление требует блокирования чтения/записи.

Пример 7. Защищенный объект блокирует задачи, вызывающие его защищенный вход. После того как количество задач в очереди ко входу достигнет пяти, объект «отпускает задачи на волю».

```
protected Blocker is
    entry Proceed;
private
     Release : Boolean := False;
end Blocker;
protected body Blocker is
    entry Proceed when
        Proceed'Count = 5 or Release is -- это барьер
    begin
        if Proceed'Count = 0 then
            Release := False;
        else
            Release := True;
        end if;
    end Proceed;
end Blocker;
```

Когда пятая задача вызывает вход `Blocker.Proceed`, она обнаружит, что барьер имеет значение `false` и поэтому задача будет блокирована. Но барьер будет перевычислен (поскольку меняется значение атрибута `'Count`). Теперь он принимает истинное значение, и первая задача в очереди обслуживается входом. Истинное значение булевой переменной `Release` гарантирует, что остальные четыре задачи тоже пройдут через барьер. После их прохода барьер снова установится в значение `false`. В течение «реализации» пяти задач встать в очередь ко входу иным задачам запрещается. Причина запрета: постановка задачи в очередь требует блокировки чтения/записи, а это невозможно обеспечить до тех пор, пока пятая задача не покинет защищенный объект.

Синхронизация процессов на основе сообщений

При организации взаимодействия и синхронизации процессов на основе сообщений выделяют два аспекта:

- модель синхронизации;
- метод именования процесса.

Вариации в модели синхронизации процесса зависят от семантики передачи сообщения. В случае асинхронного сообщения передающий процесс не ждет реакции принимающего процесса. При синхронном сообщении такая реакция ожидается.

Различают две разновидности синхронизма сообщений:

- *Рандеву* (принимающая сторона возвращает только квитанцию о приеме, после получения которой передающая сторона продолжает автономную работу).
- *Расширенное рандеву* (передающая сторона ждет квитанции и ответа как результата обработки сообщения и лишь после этого возобновляет автономную работу).

В языке Ada реализован механизм расширенного рандеву.

Метод именования процесса определяет два момента:

- вид именования (прямое/косвенное именование);
- симметрию.

При *прямом именовании* процесс-передатчик явно именует процесс-приемник:

```
send <message> to <process-name>
```

При *косвенном именовании* передатчик именует промежуточное звено (почтовый ящик):

```
send <message> to <mailbox>
```

Преимущество прямого именования — простота, косвенное именование требует введения дополнительного участника; `mailbox` может рассматриваться как интерфейс между процессами.

Схема именования *симметрична*, если и передатчик, и приемник именуют друг друга (прямо или косвенно):

```
send <message> to <process-name>;
wait <message> from <process-name>;
```

или

```
send <message> to <mailbox>;
wait <message> from <mailbox>;
```

Она асимметрична, если приемник не именует конкретный ресурс, а принимает сообщения от любого процесса (или почтового ящика):

```
wait <message>;
```

Асимметричное именование соответствует парадигме клиент-сервер.

Язык Ada поддерживает передачу сообщений между задачами, основываясь на клиент-серверной модели взаимодействия.

Задача-сервер объявляет набор услуг, которые предлагает другим задачам (своим клиентам). Для этого сервер объявляет в спецификации своей задачи публичные входы.

Каждый вход идентифицирует имя обслуживания, параметры, требуемые для приема запроса, и возвращаемые результаты:

```
entry Syn; -- параметры отсутствуют
entry Send(V : Value_Type); -- один входной параметр
entry Get(V : out Value_Type); -- один выходной параметр
entry Update(V : in out Value_Type); -- один выходной параметр
entry Mixed(A : Integer; B : out Float); -- один входной и один выходной параметр
entry Family(Boolean)(V : Value_Type); -- объявлено семейство из двух входов
    -- первый вход имеет имя Family(False)
    -- первый вход имеет имя Family(True).
```

Услугу по каждому входу поддерживает (реализует) соответствующий *оператор приема* `accept`, располагаемый в теле задачи. Например, для входа `entry Family(Boolean)(V : Value_Type)` оператор приема должен иметь следующий вид:

```
accept Family(True)(V : Value_Type) do
    -- последовательность операторов обработки
    exception -- ловушка исключений
    -- обработчики
end Family;
```

Взаимодействие задач при расширенном рандеву происходит в следующем порядке:

1. Обе задачи должны подготовиться к началу взаимодействия.
2. Если одна задача готова, а другая — нет, то готовая задача будет ждать другую.
3. Если готовы обе задачи, параметры клиента передаются в сервер.
4. Далее сервер выполняет код из оператора `accept`.
5. По окончании оператора `accept` результаты возвращаются клиенту.
6. Обе задачи освобождаются для продолжения независимой деятельности.

Пример 1. Задача-сервер — владелец магазина.

```
task Shopkeeper is
    entry Serve(X : Request; A: out Goods);
    entry Get_Money(M : Money; Change : out Money);
end Shopkeeper;
task body Shopkeeper is
begin
    loop
        accept Serve(X : Request; A: out Goods) do
            A := Get_Goods;
        end Serve;
        accept Get_Money(M : Money; Change : out Money) do
             -- взять деньги, вернуть сдачу
        end Get_Money;
    end loop;
end Shopkeeper;
```

Что неправильно в этом алгоритме взаимодействия?

Пример 2. Задача-клиент — покупатель в магазине.

```
task Customer;
task body Customer is
begin
    -- идет в магазин
```

продолжение ⇩

```
    Shopkeeper.Serve(Weekly_Shoping, Trolley); -- запрашивает товары
    -- не заплатив, в спешке покидает магазин!
end Customer;
```

Оператор `accept` может включать обработчик исключений.

ПРИМЕЧАНИЕ

Дополнительные возможности:

- Атрибут 'Count позволяет определить количество задач в очереди ко входу.
- Можно объявлять семейства входов, в действительности одноразмерный массив входов.
- Вложенные операторы accept позволяют взаимодействовать и синхронизироваться более чем двум задачам.
- При выполнении задачи внутри оператора accept может использоваться вызов другого входа.
- Исключения, не обработанные в рандеву, передаются в вызывающую и вызываемую задачи.
- Оператор accept может включать обработчик исключений.

ПРИМЕЧАНИЕ

Ограничения:

- Операторы accept могут размещаться только в теле задачи.
- Не разрешены вложенные операторы accept для одного и того же входа.
- Атрибут 'Count может быть доступен только внутри задачи, которой принадлежит вход.
- Параметры входов не могут быть параметрами-ссылками, но могут быть параметрами ссылочного типа.

Пример 3. Задача-сервер с семейством входов.

```
type Counter is (Meat, Cheese, Wine);
task CityMarket_Server is
    entry Serve(Counter)(Request: . . .);
end CityMarket_Server;

task body CityMarket_Server is
begin
    loop
        accept Serve(Meat)(. . .) do . . . end Serve; -- обслуживание по мясу
        accept Serve(Cheese)(. . .) do . . . end Serve; -- обслуживание по сыру
        accept Serve(Wine)(. . .) do . . . end Serve; -- обслуживание по вину
    end loop
end CityMarket_Server;
```

Что произойдет, если все очереди полны?

Что произойдет, если очередь к `Meat` пуста?

Пример 4. Задача-сервер с вложением одного оператора `accept` в другой.

```
task Shopkeeper is
    entry Serve_Groceries(. . .);
    entry Serve_Tobacco( . . .);
    entry Serve_Alcohol(. . .);
end Shopkeeper;
task body Shopkeeper is
begin
    …
    accept Serve_Groceries (. . .) do
        -- нет сдачи с 5000 руб., например
            accept Serve_ Alcohol(. . .) do
                -- обслуживая другого покупателя,
                -- получим больше сдачи
            end Serve_ Alcohol
    end Serve_Groceries
    …
end Shopkeeper;
```

Пример 5. Вызов входа внутри оператора `accept` для сервера запасных частей автомобиля.

```
task Car_Spares_Server is
    entry Serve_Car_Part(Number: Part_ID; . . .);
end Car_Spares_Server;
task body Car_Spares_Server is
begin
    . . .
    accept Serve_Car_Part(Number: Part_ID; . . .) do
            -- части нет в ассортименте
            Dealer.Phone_Order(. . .);
    end Serve_Car_Part;
    . . .
end Car_Spares_Server;
```

Выводы по расширенному рандеву

Основу аппарата управления рандеву составляют:

- объявление входа (`entry`);
- оператор вызова входа;
- оператор приема входа (`accept`).

Оператор вызова входа аналогичен оператору вызова простой процедуры, он помещается в задачу-клиент. Оператор приема похож на тело процедуры, но с одной оговоркой: тело процедуры может одновременно исполняться в различных асинхронных процессах, а тело оператора приема — нет.

Вызов входа `R` задачей-клиентом `K` — это заказ рандеву категории `R`, свидетельство готовности задачи-клиента `K` к рандеву с задачей-сервером `S`, в которой объявлен вход `R`. Сервер обслуживает заказ задачи `K` при достижении оператора приема (`accept`) входа `R`.

Оператор приема предписывает действия, выполняемые в ходе рандеву. При завершении этих действий обе задачи могут продолжать асинхронную работу. Если задача `S` достигла оператора приема `R` раньше, чем появился заказ `R`, то задача `S` приостанавливается до появления заказа. Таким образом, условие выполнения рандеву:

```
Готовность клиента & Готовность сервера
```

Готовность клиента означает:

```
задача K дошла до вызова входа и заказала рандеву категории R
```

Готовность сервера означает:

```
задача S дошла до оператора приема R и готова выполнить заказ
```

Рандеву заключается в том, что аргументы вызова входа `R` (из задачи-клиента) связываются с параметрами оператора приема (из задачи-сервера) и выполняется тело оператора приема, при необходимости результаты обработки посылаются из сервера в клиент. Таким образом, задачи `K` и `S` «сливаются» на время рандеву, а затем продолжают работать независимо.

«Развязка» взаимодействия задач при рандеву

Достаточно часто возникает такая ситуация, когда ожидание рандеву по одному из входов блокирует в задаче-сервере возможности обслуживания других входов. Для устранения этой неприятности служит оператор отбора `select`. Оператор отбора входов (`select`) позволяет серверу ожидать сразу несколько рандеву и отбирать (из заказанных) те рандеву, которые удовлетворяют условиям отбора, указанным в операторе `select`.

Возможны четыре формы оператора `select`:

- селективный прием `selective accept`;
- временной вызов входа `timed entry call`;
- условный вызов входа `conditional entry call`;
- асинхронный отбор `asynchronous_select`.

Селективный прием selective accept

Селективный прием позволяет серверу:

- ожидать более одного рандеву в любой момент времени;
- блокироваться по времени, если в течение заданного интервала не состоялось рандеву;
- снимать предложение к взаимодействию, если нет непосредственно доступных рандеву;
- прекратить работу, если нет клиентов, которые могут вызвать его входы.

Синтаксис этого оператора имеет следующий вид:

```
select
    [when <условие1> ⇒] <альтернатива1>;
    {or
    [when <условие2> ⇒] <альтернатива2>;}
    ...
    [else <последовательность_операторов N>]
end select;
```

где квадратные скобки обозначают необязательные элементы, а фигурные скобки — нуль и более повторений элементов.

Возможны три разновидности альтернатив:

- `<accept-оператор>; [<последовательность_операторов>;]`
- `<оператор_задержки>; [<последовательность_операторов>;]`
- `terminate; -- оператор завершения задачи.`

Порядок выполнения оператора селективного приема:

1. Вычисляются все «охраняющие» условия. Альтернативы с истинными условиями считаются открытыми.
2. Среди открытых альтернатив выбираются операторы приема с непустыми очередями вызова входов. Один из операторов приема выполняется. Конец.
3. Если нет открытых операторов приема, готовых к рандеву, то выполняется оператор задержки (`delay`). В течение задержки разрешается выполнение пункта 2. Конец.
4. Если открытая альтернатива — оператор `terminate`, то задача завершается.
5. `<Последовательность операторов N>` выполняется, если нет открытых альтернатив.

ПРИМЕЧАНИЕ

Имеются две разновидности оператора задержки:

- `delay <длительность_задержки>; -- относительная форма`
- `delay until <абсолютное_время>; -- абсолютная форма`

В оператор отбора можно включать или оператор `terminate`, или операторы `delay`, или `else`-оператор. Все три варианта взаимно исключаются. То есть в одном операторе отбора нельзя иметь альтернативу `terminate` и `else`-часть.

Пример 1. Простой отбор из двух возможных действий.

```
task Server is
    entry S1(...);
    entry S2(...);
end Server;
task body Server is
    ...
begin
    loop
        select
            accept S1(...) do
                -- код для этой услуги
            end S1;
        or
             accept S2(...) do
                -- код для этой услуги
            end S2;
        end select;
    end loop;
end Server;
```

Рассмотрим логику работы этой конструкции:

- Если нет доступных рандеву, оператор `selective accept` ждет наступления такой возможности.
- Если есть доступное рандеву, оно немедленно выбирается.
- Если есть несколько доступных рандеву, выбор зависит от реализации.
- Если в очереди к входу есть несколько задач, по умолчанию реализуется дисциплина FIFO.

Пример 2. Простой отбор из трех возможных действий.

```
type Counter is (Meat, Cheese, Wine);
task CityMarket_Server is
    entry Serve(Counter)(Request: . . .);
end CityMarket_Server ;
task body CityMarket_Server is
begin
    loop
        select
             accept Serve(Meat)(. . .) do . . . end Serve;
        or
            accept Serve(Cheese)(. . .) do . . . end Serve;
        or
            accept Serve(Wine)(. . .) do . . . end Serve;
        end select
    end loop
end CityMarket_Server ;
```

Что произойдет, если все очереди полны?

Что произойдет, если очередь к `Meat` окажется пуста?

Контрольный вопрос 1

В чем разница между двумя нижеследующими конструкциями?

```
select
    accept A;
    B;
or
    accept C;
end select;
```

и

```
select
    accept A do
        B;
    end A;
or
    accept C;
end select;
```

Напомним, что альтернативы могут иметь охраняющие условия:

- Если `охраняющее условие = true`, альтернатива имеет право быть выбранной.
- Если `охраняющее условие = false`, альтернатива не имеет права быть выбранной (при этом выполнении оператора `select`), даже если есть клиентские задачи, ждущие у входа.

Пример 3. Отбор среди охраняемых услуг.

```
type Counter is (Tobacco, Alcohol, Groceries);
task Shopkeeper is
    entry Serve(Counter)(Request: . . .);
end Shopkeeper;
task body Shopkeeper is
begin
    loop
        select
            when After_7pm => -- охрана
                accept Serve(Alcohol)(. . .) do . . . end Serve;
        or
             when Customers_Age => 16 => -- охрана
                accept Serve(Tobacco)(. . .) do . . . end Serve;
        or
            accept Serve(Groceries)(. . .) do . . . end Serve;
        end select
    end loop
end Shopkeeper;
```

Обсудим использование задержек в операторе `selective accept`.

Альтернатива `delay` в операторе `select` позволяет серверу блокироваться по времени, если в течение определенного периода не получен вызов входа. Блокировка по времени выражается с помощью оператора `delay` и может быть относительной или абсолютной.

Если относительное время имеет отрицательное значение или если абсолютное значение времени уже прошло, альтернатива `delay` становится эквивалентна альтернативе `else`. Разрешают использовать несколько задержек `delay`.

Пример 4. Рассмотрим задачу, которая каждые 10 секунд опрашивает датчики. Допустим, что в некоторых случаях может потребоваться изменение этого периода.

```
task Sensor_Monitor is
    entry New_Period(P : Duration);
end Sensor_Monitor;
task body Sensor_Monitor is
    Current_Period : Duration := 10.0;
    Next_Cycle : Time := Clock + Current_Period;
begin
    loop
        -- чтение значения датчика и т. д.
        select
            accept New_Period(P : Duration) do
                Current_Period := P;
            end New_Period;
            Next_Cycle := Clock + Current_Period;
        or
            delay until Next_Cycle; -- альтернатива delay
            Next_Cycle := Next_Cycle + Current_Period;
        end select;
    end loop;
end Sensor_Monitor;
```

Пример 5. Использование `else`-части в операторе `selective accept`.

```
task body Sensor_Monitor is
    Current_Period : Duration := 10.0;
    Next_Cycle : Time := Clock + Current_Period;
begin
    loop
            -- чтение значения датчика и т. д.
         select
            accept New_Period(P : Duration) do
                Current_Period := P;
            end New_Period;
            else -- не может охраняться Часть else
                null;
         end select;
        Next_Cycle := Clock + Current_Period;
        delay until Next_Cycle;
    end loop;
end Sensor_Monitor;
```

В этом случае оператор `select` не допускает никакого ожидания рандеву: при отсутствии запроса клиента сразу срабатывает `else`-часть.

Напомним, что в одном операторе `select` нельзя смешивать `else`-часть и `delay`.

Следующие формы эквивалентны:

```
select
    accept A;
or
    accept B;
else
    C;
end select;
select
    accept A;
or
    accept B;
or
    delay 0.0;
    C;
end select;
```

Контрольный вопрос 2

В чем разница между тремя нижеследующими конструкциями?

```
select
    accept A;
or
    delay 10.0;
end select;

select
    accept A;
else
    delay 10.0;
end select;

select
    accept A;
or
    delay 5.0;
    delay 5.0;
end select;
```

Временной вызов входа

Временной вызов входа — это такой вызов, который отменяется, если вызов не принят в течение определенного периода (относительного или абсолютного). В нем можно определить только одну альтернативу задержки и только один вызов входа.

Пример 6. Покупатель, которого надо обслужить за 10 секунд.

```
task type Shopper;
task body Shopper is
begin
    . . .
    -- вошел в магазин
    select
        shopkeeper.Serve_Groceries(. . .)
    or
        delay 10.0;
        -- пожаловался на очереди;
    end select;
    -- покинул магазин
    . . .
end Shopper;
```

Условный вызов входа

Условный вызов входа позволяет клиенту отказаться от взаимодействия, если задача-сервер не готова принять его вызов немедленно. Тот же смысл имеет временной вызов входа с нулевым сроком задержки.

Пример 7. Немедленное включение света.

```
select
    Security_Op.Turn_Lights_On;
else
    null; -- предполагается, что свет уже включен
end select;
```

Условный вызов входа должен использоваться тогда, когда в случае непринятия вызова задача может выполнять другую полезную работу.

Заметим, что условный вызов входа использует `else`, а временной вызов входа использует `or`. Эти два типа вызовов смешивать нельзя.

Асинхронный отбор

Оператор асинхронного отбора обеспечивает асинхронную передачу управления, имеет вид:

```
select
    delay 5.0;-- переключающая альтернатива
    Put_Line("Вычисление не закончено");
then abort
    Обратить_Большую_Матрицу(M); -- часть останова
end select;
```

Порядок выполнения: если операторы из части останова не выполнены до истечения срока задержки, то они прекращают работу. Таким образом, если матрица

не обращена за 5 секунд, получим донесение «Вычисление не закончено». Вместо оператора задержки может использоваться вызов входа. Если вызов исполняется до завершения части останова, тогда вычисления тоже прекращаются. Выполняются операторы, следующие за вызовом входа. С другой стороны, если вычисления завершаются до вызова входа, тогда вызов входа сам останавливается.

Потоки языка Java

Параллельными модулями в языке Java являются объекты, содержащие метод `run()`, код которого выполняется параллельно с подобными методами из других объектов и методом `main()`. Процесс, в котором выполняется любой метод `run()`, здесь называется потоком. Потоки в Java представляют собой облегченные задачи, они запускаются в одном и том же адресном пространстве. Этим потоки отличаются от задач в Ada, которые считаются тяжеловесными процессами, требующими значительно больших ресурсов.

Есть две возможности определить класс с методом `run()`. Можно объявить производный класс от предопределенного класса `Thread` и переопределить его метод `run()`. Однако если у нашего класса уже есть родитель, этот прием не работает. В подобных ситуациях определяют подкласс, который наследует его характеристики и реализует интерфейс `Runnable`. `Runnable` обеспечивает только протокол метода, так что реализующий класс должен предоставить полное определение `run()`. Объект класса, реализующего `Runnable`, передается в конструктор класса `Thread`. Следовательно, без объекта класса `Thread` не обойтись и в этом случае.

В языке Ada задачи могут быть как клиентами, так серверами, взаимодействующими друг с другом посредством операторов `accept`. Методы `run()` в Java являются клиентами, и нет механизма для их взаимных коммуникаций, за исключением метода `join()` и разделяемых переменных.

Класс Thread

Класс `Thread` не может быть естественным родителем для любых других классов. Он лишь предлагает подклассам некоторые услуги, но никак не связан с теми вычислительными функциями, которые должны нести в себе эти подклассы. И тем не менее класс `Thread` — это единственный класс, доступный программисту для создания параллельных программ на языке Java.

Класс `Thread` содержит пять конструкторов и набор методов и констант. Метод `run()`, описывающий действия в потоке, всегда переопределяется подклассами `Thread`. Метод `start()` класса `Thread` вызывает метод `run()`, инициируя новый поток как параллельный модуль. Необычность вызова `start()` заключается в том, что управление возвращается непосредственно вызывающему модулю, который продолжает свою работу параллельно стартовавшему методу `run()`.

Запуск метода в новом потоке обеспечивается следующим фрагментом:

```
class TheThread extends Thread {
    public void run() { . . . }
}
```

```
. . .
Thread concObj = new TheThread();
concObj.start();
```

Когда прикладная программа на языке Java начинает свое выполнение, создается новый поток, в котором будет выполняться метод `main()`, и вызывается сам метод `main()`. Таким образом, все программы на языке Java выполняются с помощью потоков.

Когда программа имеет множество потоков, планировщик должен определить порядок выполнения потоков. Чаще всего доступен лишь один процессор, поэтому потоки выполняются по одному. Обычно, если потоки имеют одинаковый приоритет, планировщик циклически предоставляет одинаковые отрезки времени каждому запускаемому потоку.

Класс `Thread` имеет несколько методов для управления выполнением потоков. Метод `yield()`, не имеющий параметров, фиксирует отказ выполняемого потока от остатка выделенного времени. Поток немедленно помещается в очередь задач, готовых к выполнению, становясь вновь запускаемым. Далее планировщик выбирает из очереди задач, готовых к выполнению, поток с наивысшим приоритетом. Если в очереди нет потоков с более высоким приоритетом, чем у потока, только что освободившего процессор, этот поток может сразу же получить процессорное время.

Метод `sleep()` имеет один параметр — количество миллисекунд, на которое вызывающий модуль хочет заблокировать поток (перевести в состояние сна). После того как истечет указанный интервал, поток будет помещен в очередь задач, готовых к выполнению. Поскольку невозможно узнать, как долго поток будет пребывать в очереди задач, готовых к выполнению, прежде чем он начнет свое выполнение, параметр метода `sleep()` указывает минимум времени, в течение которого поток не будет выполняться. Метод `sleep()` генерирует исключение `InterruptedException`, которое должно обрабатываться методом, вызвавшим метод `sleep()`.

Метод `join()` запрашивает задержку на выполнение метода. Задержка запрашивается до того момента, пока метод `run()` другого потока не завершит свою работу. Метод полезен в ситуации, когда дальнейшая обработка не может быть продолжена без завершения работы другого потока. Допустим, что имеется следующий метод `run()`:

```
public void run() {
    . . .
    Thread concObj = new TheThread ();
    concObj.start();
    // выполнить часть вычислений с помощью этого потока
    concObj.join(); // ждать завершения concObj
    // выполнить остаток вычислений этого потока
}
```

Метод `join()` помещается в поток, вызывающий собственную блокировку до момента завершения потока, к которому применяется `join()`. Если поток заблокирован, возникает опасность тупика (взаимной блокировки). Для предотвращения подобного эффекта `join()` может вызываться с параметром, который ограничивает (в миллисекундах) время ожидания завершаемого потока. Например, вызов:

```
concObj.join(1000);
```

ограничит ожидание завершения `concObj` одной секундой. Если завершение не состоялось (в отведенное время), вызывающий поток возвращается в очередь задач, готовых к выполнению. Это значит, что его выполнение будет продолжено по мере возможности.

В ранних версиях языка Java класс `Thread` включал три дополнительных метода: `stop()`, `suspend()` и `resume()`. Все они признаны устаревшими из-за проблем безопасности. К примеру, метод `stop()` иногда переопределялся простым методом, который уничтожал поток путем установки его ссылочной переменной в `null`.

Обычно запуск метода `run()` завершается после выполнения всех его операторов. Иногда поток нужно уведомить о необходимости досрочного прекращения. Самый простой способ такого уведомления: послать в поток сообщение, которое устанавливает нужное значение проверяемого бита потока. Для проверки используется метод `isInterrupted()`. Это частичное решение, поскольку поток может оказаться в состоянии задержки или блокировки, в котором проверка не выполняется. В подобных ситуациях метод прерывания генерирует исключение `InterruptedException`, пробуждающее поток (от сна или ожидания). Следовательно, поток периодически проверяет бит прерывания и не может пропустить эту проверку.

Приоритеты

Приоритеты у потоков могут быть разными. По умолчанию поток имеет тот же приоритет, что и создавший его поток. Если поток создан методом `main()`, его приоритет по умолчанию равен константе `NORM_PRIORITY`, которая по умолчанию равна 5. В классе `Thread` определены две другие константы, задающие приоритеты `MAX_PRIORITY` и `MIN_PRIORITY`, со значениям 10 и l соответственно. Приоритет потока можно изменить с помощью метода `setPriority()`. Новый приоритет может быть равен любой из заранее определенных констант или любому целому числу между значениями `MIN_PRIORITY` и `MAX_PRIORITY`. Метод `getPriority()` возвращает текущий приоритет потока.

При наличии потоков с различными приоритетами работа планировщика диктуется этими приоритетами. Если выполнение текущего потока заблокировано или прекращено либо истек отведенный интервал времени, планировщик выбирает в очереди задач, готовых к выполнению, поток с наивысшим приоритетом. Поток с наименьшим приоритетом запускается лишь тогда, когда нет потока с более высоким приоритетом.

Семафоры

В пакете `java.util.concurrent.Semaphore` находится определение класса `Semaphore`. Объекты этого класса реализуют счетные семафоры. В классе определены два метода `acquire()` и `release()`, которые соответствуют рассмотренным нами операциям `wait()` и `signal()`.

Базовый конструктор класса `Semaphore` принимает целый параметр, который инициализирует счетчик семафора. Например, инициализируем семафоры для доступа к ограниченному буферу:

```
itemAvailable = new Semaphore(0);
spaceAvailable = new Semaphore(size);
```

Операцию для поставщика можно представить в следующем виде:

```
spaceAvailable.acquire();
append(value);
itemAvailable.release();
```

Соответственно, операция для потребителя принимает вид:

```
itemAvailable.acquire();
take(value);
spaceAvailable.release();
```

Синхронизация конкуренции

В языке Java методы (не конструкторы) можно определить как синхронизированные. Вызванный в конкретном объекте синхронизированный метод должен завершить свою работу, прежде чем в этом объекте может быть запущен другой синхронизированный метод. Синхронизация конкуренции над объектом организуется за счет синхронизации методов для доступа к разделяемым данным. Механизм синхронизации реализован следующим образом: каждый объект в Java имеет блокировку. Синхронизированный метод должен заблокировать объект перед началом своей работы, это предотвращает возможность выполнения объектом (в этот период) других синхронизированных методов. Синхронизированный метод снимает блокировку с объекта после окончания своей работы. Рассмотрим следующий фрагмент:

```
class Buffer {
    private int [100] buf;
    . . .
    public synchronized void append(int element) { . . . }
    public synchronized int take() { . . . }
    . . .
}
```

Оба метода, определенных в классе `Buffer`, являются синхронизированными, что предотвращает их взаимное влияние во время работы с одним и тем же объектом, даже если они вызываются разными потоками.

Объект, в котором все методы синхронизированы, по сути, является монитором. В общем случае объект может содержать и синхронизированные, и несинхронизированные методы. Несинхронизированный метод может запускаться в объекте в любое время, даже параллельно выполнению синхронизированного метода.

В некоторых случаях количество операторов для обработки разделяемых данных существенно меньше общего числа операторов в теле метода. В подобной ситуации целесообразно синхронизировать лишь фрагмент тела, а не весь метод. Такая конструкция называется синхронизированным оператором:

```
synchronized (выражение){
    операторы
}
```

где выражение должно вычислять объект, а оператор может быть как простым, так и составным. На время выполнения этого оператора или операторов объект

блокируется. Таким образом, оператор или операторы выполняются точно так же, как если бы они были телом синхронизированного метода.

Объект, для которого определены синхронизированные методы, должен иметь связанную с ним очередь, хранящую запросы синхронизированных методов, приходящиеся на тот период, когда объект был блокирован. Такую структуру называют *очередью внутреннего состояния*. Она имеется у любого объекта и обслуживается неявно. При разблокировании объекта запрос на метод, находящийся в очереди внутреннего состояния, помещается в очередь задач, готовых к выполнению.

Синхронизация взаимодействия

Синхронизация взаимодействия в языке Java осуществляется с помощью методов `wait()`, `notify()` и `notifyAll()`, определенных в классе `Object`, корневом классе для всех классов языка Java. Все классы наследуют эти методы.

Метод `wait()` всегда блокирует вызывающий поток, он позволяет приостановить выполнение потока до того момента, пока не будет удовлетворено определенное условие. Каждый метод имеет список ожиданий всех потоков, которые вызывали `wait()` над объектом.

Метод `notify()` пробуждает один ожидающий поток, он предназначен для того, чтобы сообщить ожидающему потоку, что ожидаемое событие произошло. Достаточно трудно выявить конкретный поток, который пробуждается методом `notify()`, поскольку виртуальная машина Java (JVM) выбирает его случайным образом. В силу этого чаще всего используют метод `notifyAll()`. Этот метод пробуждает все потоки из объектного списка ожидания, помещая их в очередь задач, готовых к выполнению.

Методы `wait()`, `notify()` и `notifyAll()` можно вызвать только внутри синхронизированного метода, поскольку они используют блокировку, наложенную на объект таким методом. Вызов `wait()` всегда помещается в цикл, управляемый условием, которое ожидается методом. Для `notify()` или `notifyAll()` цикл желателен, если ожидаемое условие может поменяться.

Метод `wait()` может возбудить исключение `InterruptedException`, являющееся потомком исключения `Exception`. Следовательно, любой код, вызывающий метод `wait()`, должен также перехватывать исключение `InterruptedException`. Предположим, что условие, выполнения которого мы ожидаем, называется `myCond`. Тогда метод `wait()` удобно использовать следующим образом:

```
try {
    while (!myCond)
    wait();
    -- Выполнить все необходимое после установки истинного значения myCond
}
catch(InterruptedException theP) { . . . }
```

Следующая программа реализует очередь для хранения значений типа `long`. Она иллюстрирует синхронизацию как взаимодействия, так и конкуренции.

```
class Queue {
    private long [] queArray;
    private int rear, front, nItems, maxSize;
    public Queue(int s) {
        queArray = new long[s];
```

```
        nItems = 0; rear = -1;
        front = 0; maxSize = s;
    } //** окончание конструктора
    public synchronized void append (long item)
        throws InterruptedException {
            try {
                while (nItems == maxSize)
                    wait();
                    queArray[++rear] = item;
                    nItems++;
                    notifyAll();
                } //** end of try clause
            catch(InterruptedException e) {}
        } //** окончание метода append
    public synchronized long take()
        throws InterruptedException {
            long item = 0;
            try {
            while (nItems == 0)
                wait();
            item = queArray[front++];
            nItems--;
            notifyAll();
        } //** end of try clause
        catch(InterruptedException e) {}
        return item;
    } //** окончание метода take
} //** окончание класса Queue
```

Потоковый класс-производитель, а также потоковый класс-потребитель, работающие с объектом класса Queue в режиме взаимного исключения, можно определить в следующем виде:

```
class Producer extends Thread {
    private Queue monitor;
    public Producer(Queue buf) {
        monitor = buf;
    }
    public void run() {
        long intoMonitor;
        while (true) {
            //--Произвести новый элемент
            monitor.append(intoMonitor);
        }
    }
}
class Consumer extends Thread {
    private Queue monitor;
    public Consumer(Queue buf) {
        monitor = buf;
    }
    public void run() {
        long fromMonitor;
        while (true) {
            fromMonitor = monitor.take();
            //-- Потребить существующий элемент
        }
    }
}
```

Следующий фрагмент программы запускает потоки производителя и потребителя:

```
Queue mon = new Queue(5000);
Producer prod = new Producer(mon);
Consumer cons = new Consumer(mon);
prod.start();
cons.start();
```

Альтернативное определение потокового класса предполагает реализацию интерфейса `Runnable`, а не прямое наследование от класса `Thread`. Например, класс `Producer` можно записать так:

```
class Producer implements Runnable {
    private Queue monitor;
    public Producer(Queue buf) {
        monitor = buf;
    }
    public void run() {
        long intoMonitor;
        while (true) {
            //--Произвести новый элемент
            monitor.append(intoMonitor);
        }
    }
}
```

Как видим, различие проявляется лишь в первой строке определения. Правда, меняется логика создания и запуска соответствующего объекта (приходится объявлять объект класса `Thread`, связанный с объектом-производителем):

```
Producer prod2 = new Producer(mon);
Thread prodThread = new Thread(prod2);
prodThread.start();
```

Здесь объект класса `Queue` передается в конструктор класса `Producer`, а экземпляр класса `Producer` передается в конструктор класса `Thread`.

Неблокирующая синхронизация

Язык Java содержит несколько классов для такого управления доступом к переменным, который не включает блокирования или ожидания. Пакет `java.util.concurrent.atomic` определяет классы, разрешающие неблокирующий синхронизированный доступ к переменным простого типа `int`, `long` и `Boolean`, а также к ссылкам и массивам. Например, класс `AtomicInteger` определяет методы чтения и записи, а также методы для сложения, инкремента и декремента. Все эти операции *атомарны*, то есть не могут прерываться, следовательно, для обеспечения целостности переменных в многопоточных программах блокировка не нужна. Этой «мелкозернистой» синхронизации достаточно для простых переменных. Большинство процессоров содержит атомарные команды для этих операций над целыми значениями, поэтому они просты для реализации и не требуют неявной блокировки.

Неблокирующая синхронизация очень эффективна. Неблокирующий доступ в конкурентной борьбе за ресурсы по определению быстрее синхронизированного доступа, поскольку последний требует приостановки и реструктуризации потоков.

Явная блокировка

В версии 5.0 язык Java вводит явную блокировку как альтернативу синхронизированному методу и блокам, обеспечивающим неявную блокировку. Интерфейс `Lock` объявляет методы `lock()`, `unlock()` и `tryLock()`. Предопределенный класс `ReentrantLock` реализует интерфейс `Lock`. Для блокировки программного блока используют следующую идею:

```
Lock lock = new ReentrantLock();
. . .
Lock.lock();
try {
    // Код для доступа к разделяемым данным
} finally {
    Lock.unlock();
}
```

Здесь создается объект класса `Lock`, в котором вызывается метод `lock()`. Критический код заключается в блок `try`. Вызов метода `unlock()` в секции `finally` гарантирует, что блокировка будет снята, вне зависимости от того, что происходило в блоке `try`.

Существуют, по меньшей мере, две ситуации, в которых явную блокировку предпочитают неявной. Во-первых, если необходимо получение блокировки в течение короткого интервала времени. Для этой ситуации интерфейс `Lock` предлагает метод `tryLock()`, параметр которого задает ограничение по времени. Если в течение этого времени блокировка не предоставлена, передается управление оператору, следующему за вызовом `tryLock()`. Во-вторых, если неприемлема пара «блокировать-разблокировать», оформленная в виде структурированного блока. При неявном подходе разблокирование всегда выполняется в конце того составного оператора, начало которого сопровождалось блокированием. При явной блокировке разблокирование может производиться где угодно, вне зависимости от структуры программы.

Главная опасность явной блокировки — пропуск процедуры разблокирования (при неявной блокировке подобный пропуск невозможен). В этом случае приложение рискует оставаться заблокированным навечно.

Как уже указывалось, каждому объекту придается специальная структура — очередь внутреннего состояния, которая запоминает запросы потоков, ожидающих некоторого условия, связанного с объектом. Интерфейс этой очереди образуют методы `wait()`, `notify()` и `notifyAll()`. Поскольку каждый объект может иметь только одну очередь состояния, в ней могут находиться запросы потоков, ожидающие исполнения самых различных условий. Например, в ранее приведенном примере для «производителя-потребителя» потоки могут ожидать в экземпляре класса `Queue` исполнения одного из двух условий (`nItems == maxSize` или `nItems == 0`). По этой причине монитор использует метод `notifyAll()`. Если применить `notify()`, то пробуждается только один поток, а ведь он может ожидать другого условия (не того, которое получило истинное значение). Использование метода `notifyAll()` — дорогое удовольствие, поскольку он пробуждает все потоки, а все они должны проверить свои собственные условия, чтобы определиться с запуском. Более того, для проверки их условий потоки прежде всего запрашивают блокировку объекта.

Альтернативой очереди внутреннего состояния считается интерфейс `Condition`, который использует очередь состояния, связанную с объектом класса `Lock`. Он также объявляет альтернативы для `wait()`, `notify()` и `notifyAll()`, именуемые как `await()`, `signal()` и `signalAll()`. С каждым объектом класса `Lock` может быть связано любое количество объектов класса, реализующего `Condition`. В контексте `Condition` метод `signal()` предпочтительнее метода `signalAll()`, поскольку он легче для понимания и более эффективен (приводит к меньшему количеству переключений контекста).

Потоки в C#

Потоки в языке C# создавались «по мотивам» потоков языка Java. Тем не менее между ними имеются существенные различия.

Основные операции над потоками

В языке Java для запуска потока используют единственный метод `run()`; в C# любой метод может быть запущен в своем собственном потоке. При создании поток в C# связывается с экземпляром предопределенного делегата `ThreadStart`. Когда выполнение потока начинается, его делегат имеет адрес метода, который он должен запустить. Таким образом, выполнение потока контролируется с помощью связанного делегата.

Поток в C# формируется путем создания объекта класса `Thread`. В конструктор класса `Thread` должна быть передана конкретизация `ThreadStart`, которой, в свою очередь, должно быть передано имя метода, запускаемого в потоке. Рассмотрим пример:

```
public void TheRun() { . . . }
. . .
Thread theThread = new Thread(new ThreadStart(TheRun));
```

В этом примере создается поток с именем `theThread`, чей делегат указывает на метод `TheRun`. Таким образом, когда поток начинает выполнение, он вызывает метод, чей адрес находится в его делегате. В данном примере `theThread` является делегатом, а `TheRun` — методом.

В языке C# существует две категории потоков: актеры и серверы. Потоки-актеры не вызываются специально, скорее они просто стартуют. Кроме того, методы, которые они выполняют, не принимают параметры и не возвращают значения. Как и в языке Java, создание потока не приводит к старту его параллельного выполнения. Для потоков-актеров выполнение должно запрашиваться с помощью метода `Start()` класса `Thread`:

```
theThread.Start();
```

Как и в языке Java, можно обеспечить, чтобы поток для своего продолжения ожидал завершения другого потока. Для этого используется метод `Join()`. К примеру, допустим, что поток `X` включает в себя следующий вызов:

```
Y.Join();
```

Поток X будет блокирован до тех пор, пока существует поток Y.

Метод Join() может принимать параметр типа int, определяющий ограничение по времени (в миллисекундах), в течение которого вызывающий модуль ждет завершения потока.

Поток можно приостановить на определенный промежуток времени, для этого используется публичный статический метод Sleep() класса Thread. Параметром Sleep() является целое число в миллисекундах. В отличие от аналога в языке Java, здесь Sleep() не генерирует никаких исключений, поэтому может вызываться вне блока try.

Поток может быть завершен с помощью метода Abort(), хотя это не приводит к «убийству» в буквальном смысле слова. На самом деле метод Abort() генерирует исключение ThreadAbortException, которое поток может перехватить. Когда поток перехватывает это исключение, обычно он освобождает задействованные ресурсы, а затем останавливается (добравшись до конца собственного кода).

Поток-сервер запускается путем вызова его делегата. Такие потоки называются серверами, поскольку они предоставляют по запросу некую услугу. Потоки-серверы интереснее потоков-актеров, так как обычно взаимодействуют с другими потоками. Достаточно часто их выполнение синхронизируется с другими потоками.

Повторим, что в C# любой метод можно вызвать косвенно, с помощью делегата. Такие вызовы указываются с использованием объекта делегата, выступающего в роли имени метода. На самом деле речь идет о сокращенной записи вызова метода делегата Invoke(). К примеру, если делегат имеет имя myMeth, а метод принимает один целый параметр, то вызов метода можно записать в одной из двух следующих форм:

```
myMeth (75);
myMeth.Invoke(75);
```

Такие вызовы считаются синхронными; это значит, что при вызове метода вызывающий модуль блокируется до тех пор, пока вызванный метод не завершит свое выполнение. Кроме того, язык C# поддерживает асинхронные вызовы методов, выполняемых в потоках. Когда поток вызывается асинхронно, вызываемый и вызывающий потоки выполняются параллельно, поскольку вызывающий не блокируется на время работы вызванного потока.

Поток вызывается асинхронно посредством экземплярного метода делегата с именем BeginInvoke(), в который передаются параметры для метода делегата, а также два дополнительных параметра: один типа AsyncCallback, другой — типа Object. Метод BeginInvoke() возвращает объект, реализующий интерфейс IAsyncResult. Кроме того, класс делегата определяет экземплярный метод EndInvoke(), принимающий один параметр типа IAsyncResult и возвращающий результат того типа, который задан для метода, инкапсулированного в объекте делегата. Итак, для асинхронного вызова потока используется BeginInvoke(). В следующем примере для двух последних параметров зададим значение null. Предположим, что имеется следующее объявление метода и определение потока:

```
public float TheMethod(int a);
. . .
Thread theThread = new Thread(new ThreadStart(TheMethod));
```

Тогда асинхронный вызов TheMethod() записывается в форме:

```
IAsyncResult total = theThread.BeginInvoke(25, null, null);
```

Возвращаемое значение вызванного потока обеспечивается методом `EndInvoke()`, принимающим в качестве параметра объект (типа `IAsyncResult`), который возвращается методом `BeginInvoke()`. Метод `EndInvoke()` возвращает результат вызванного потока. К примеру, для получения вещественного результата вызова метода `TheMethod()` нужно использовать следующий оператор:

```
float returnResult = EndInvoke(total);
```

Если вызывающий поток продолжает работу параллельно выполнению вызванного потока, должен быть способ сигнализации о том, что вызванный поток финишировал. С этой целью интерфейс `IAsyncResult` определил свойство `IsCompleted`. В то время как вызванный поток выполняется, действия вызывающего потока могут определяться операторами, вставленными в цикл `while`, который зависит от условия `IsCompleted`. Например:

```
IAsyncResult total = theThread.BeginInvoke(25, null, null);
while(!total.IsCompleted) {
    // Какие-нибудь вычисления
}
```

С одной стороны, достаточно удобно что-то вычислять в вызывающем потоке в ожидании того момента, когда вызванный поток завершит свою работу. С другой стороны, если количество вычислений в цикле сравнительно мало, то подавляющая часть времени расходуется лишь на проверку условия `IsCompleted`. Для уменьшения «холостого пробега» можно прописать в вызывающем потоке делегата с адресом метода обратного вызова. Метод обратного вызова выполняется по завершению вызванного потока. Подобный делегат передается в метод `BeginInvoke()` как второй параметр. Приведем пример соответствующего вызова `BeginInvoke()`:

```
IAsyncResult total = theThread.BeginInvoke(25,
    new AsyncCallback(TheMethodComplete), null);
```

Метод обратного вызова определяется в вызывающем потоке. Такие методы просто устанавливают булеву переменную, например `isDone`, в значение `true`. Независимо от времени работы вызванного потока метод «обратного вызова» вызывается лишь один раз.

Синхронизация потоков

Существуют три различных способа синхронизации потоков в C#: класс `Interlocked`, класс `Monitor` из пространства имен `System.Threading` и оператор `lock`. Каждый из этих механизмов создан с определенной целью. Класс `Interlocked` применяется для синхронизации операций по инкрементированию и декрементированию целых чисел. Эти операции автоматически выполняются с помощью двух методов класса `Interlocked` с именами `Increment()` и `Decrement()`, которые в качестве параметра используют ссылку на целую величину. К примеру, для инкрементирования в потоке разделяемой целой переменной по имени `sum` можно использовать:

```
Interlocked.Increment(ref sum);
```

Оператор `lock` применяется для того, чтобы пометить в потоке критическую секцию программного кода. Проиллюстрируем его использование:

```
lock(sign) {
    // Критическая секция
}
```

Если нужно синхронизировать код в приватном экземплярном методе, а `sign` — это текущий объект, то он и используется как знак для блокировки. Если же синхронизируемый код находится в публичном экземплярном методе, то создается новый экземпляр класса `Object` (в классе метода с синхронизируемым кодом), а ссылка на него применяется в качестве знака для блокировки.

Класс `Monitor` определяет пять методов `Enter()`, `Wait()`, `Pulse()`, `PulseAll()` и `Exit()`, которые обеспечивают управление синхронизацией потоков. Метод `Enter()`, принимающий ссылку на объект как параметр, помечает начало синхронизации потока для данного объекта. Метод `Wait()` приостанавливает выполнение потока и оповещает систему поддержки вычислений .NET о том, что данный поток хочет возобновить свое выполнение в следующий раз (по мере возможности). Метод `Pulse()`, принимающий ссылку на объект как параметр, уведомляет о возможности повторного запуска для одного ожидающего потока. Метод `PulseAll()` подобен методу `notifyAll()` в языке Java. Ожидающие потоки запускаются в том порядке, в котором они вызывались методом `Wait()`. Метод `Exit()` завершает работу критической секции потока.

В итоге компиляции оператор `lock` превращается в монитор. Монитор используется при необходимости дополнительного управления (например, совместно с `Wait()` и `PulseAll()`).

Среда .NET 4.0 ввела набор родовых параллельных структур данных для очередей, стеков и `bags` (так называют неупорядоченные наборы объектов). Эти новые потоковые классы считаются безопасными в контексте распараллеливания, то есть в многопотоковых программах они не инициируют конкуренцию при синхронизации.

Вышеперечисленные классы определены в пространстве имен `System.Collections.Concurrent` под именами `ConcurrentQueue<T>`, `ConcurrentStack<T>` и `ConcurrentBag<T>`. Приложение с производителем и потребителем на языке C# легко написать на основе `ConcurrentQueue<T>`, при этом нет нужды заботиться о синхронизации конкуренции.

Контрольные вопросы и упражнения

1. Реализуйте механизм семафора на языке Ada. Напишите программу с двумя параллельными процессами, в которой семафор обеспечивает режим взаимного исключения.
2. На языке Ada напишите программу с двумя параллельными процессами, в которой семафор обеспечивает условную синхронизацию.
3. Реализуйте механизм монитора на языке Ada. Напишите программу с двумя параллельными процессами, в которой используется монитор.
4. Положим, что режим широковещания поддерживается следующим типом защищенного объекта:

```
protected type Broadcast is
    entry Receive(M : out message);
    procedure Send(M : message);
private
    New_Message : Message;
    Message_Arrived : Boolean := False;
end Broadcast;
```

Каждый из очереди к `Receive` должен получить сообщение, посылаемое при вызове `send`. Напишите реализацию этого типа и прикладную программу, использующую его.

5. На языке Ada напишите программу, реализующую модель мультиплексора на четыре входа с использованием задач.
6. Возможно ли реализовать операции `wait` и `signal` семафора с помощью сообщений `send` и `receive` из четвертого упражнения? Напишите две процедуры `signal(P)` и `wait(P)`, которые взаимодействуют с помощью сообщений, но семантически имеют тот же смысл, что и операции семафора `wait` и `signal`. Какие проблемы с атомарностью возникают при таком решении? Операции `wait` и `signal` могут использоваться множеством задач, применяющих один и тот же семафор. Как можно обобщить сообщения `send` и `receive`, чтобы они могли использоваться более чем одной посылающей и одной принимающей задачей?

Глава 20

Функциональное программирование

В этой главе излагаются вопросы, связанные с функциональным подходом к написанию программ. Обсуждаются специфические особенности функциональных языков программирования. На примере языка Scheme рассматриваются классические понятия и средства функционального программирования: выражения, специальные формы, создание функций, динамическая проверка типа, хвостовая и не хвостовая рекурсия, структуры данных, специфика приемов программирования, функции высшего порядка, статическая область видимости, настройка — специализация языка. Далее на примере языка ML освещается функциональный подход со статической типизацией, подчеркиваются концептуальные вопросы, которые приходится при этом решать. Здесь же поясняется понятие карризации функций, приводятся примеры создания карризованных функций. В заключение описывается проблематика отложенных вычислений: нестрогие функции, специальные формы для отложенных вычислений, передача по необходимости, мемоизация, ленивые вычисления.

Особенности функциональных языков программирования

Программа представляет собой описание конкретных вычислений. Если отвлечься от деталей вычислений — «как» вычислять — и сосредоточиться на вычисляемом результате — «что» вычислять, — то программа становится виртуальным черным ящиком, который преобразует входные данные в выходные. С этой точки зрения программа по существу эквивалентна математической функции.

ОПРЕДЕЛЕНИЕ. Функция является правилом, которое каждому элементу `x` из некоторого множества определения `X` ставит в соответствие единственный элемент `y` из множества значений `Y`. В математических терминах, если `f` — имя функции, мы можем записать:

```
y = f(x)
```

или

```
f: X → Y
```

Множество `X` называют *областью определения* функции `f`, а множество `Y` — *областью значений* функции `f`. Величина `x` в `f(x)`, которая представляет собой любой элемент из множества `X`, называется независимой переменной, а величину `y` из множества `Y`, определяемую уравнением `y = f(x)`, называют зависимой переменной. Иногда, если функция `f` не определена для всех `x` в `X`, говорят о частично определенной функции, в противном случае имеют в виду полное определение.

Очень важное свойство математического определения функции состоит в том, что при заданном аргументе `x` она всегда определяет одно и то же значение, поскольку не имеет побочных эффектов. Побочные эффекты в языках программирования связаны с переменными, моделирующими ячейки памяти. Математическая функция определяет значение, а не указывает последовательность операций над числами, хранящимися в ячейках памяти, для вычисления некоторого значения. Здесь нет переменных в том смысле, который придается им в императивных языках программирования, поэтому не может быть побочных эффектов.

В языке программирования на основе математического понятия функции могут быть представлены и программы, и процедуры, и функции. В случае программы `x` соответствуют входные данные, а `y` — результаты. В случае процедуры или функции `x` характеризует параметры, а `y` — возвращаемые значения. В любом случае на `x` можно ссылаться как на «вводы», а на `y` — как на «выводы». Следовательно, в функциональном подходе к программированию не делают различий между программой, процедурой и функцией. С другой стороны, входные и выходные величины всегда различаются.

В языках программирования совершенно четко разделены определение функции и применение функции. *Определение функции* описывает, как можно вычислить величину на основе формальных параметров. *Применение функции* заключается в вызове конкретной функции с использованием фактических параметров. Заметим, что в математике разница между параметром и переменной не всегда очевидна. Очень часто термин «параметр» заменяют термином «независимая переменная». К примеру, в математике можно записать определение функции возведения в квадрат:

```
square(x) = x * x
```

Далее функция частенько применяется к переменной `x`, представляемой фактическим значением:

```
Положим, что для x выполняется square(x) + 2 . . .
```

Основное различие между императивным и функциональным программированием состоит в *интерпретации понятия переменной* [3, 16, 31, 32, 36, 55, 64, 74, 75, 86, 88]. В математике переменные представляются как фактические значения, а в императивных языках программирования переменные ссылаются на области памяти, где хранятся их значения. Изменить значения в этих областях памяти позволяют присваивания. Напротив, в математике нет понятий «область памяти» и «присваивание», поэтому такой оператор, как

```
x = x + 1
```

не имеет смысла. Функциональное программирование основывается на математическом подходе к понятию «переменная». В функциональном программировании переменные связаны со значениями, но не имеют никакого отношения к областям памяти. После привязывания переменной к значению значение переменной меняться

не может. Это исключает необходимость в операции для изменения значения переменной, то есть в присваивании.

Несмотря на то что большинство функциональных языков программирования поддерживает ту или иную форму присваивания, в них обеспечивается возможность создания чисто функциональной программы, основанной на строго математическом подходе к переменным.

В силу отсутствия присваивания циклы в функциональном программировании просто невозможны, поскольку цикл нуждается в управляющей переменной, чье значение должно изменяться по мере выполнения цикла. Следовательно, единственной формой для записи повторяющихся операций становится рекурсия. К примеру, определим императивную форму вычисления наибольшего общего делителя (greatest common divisor — gcd) на основе цикла:

```
void gcd( int u, int v, int *x)
{   int y, t, z;
    z = u ; y = v;
    while (y != 0)
        {   t = y;
            y = z % y;
            z = t;
        }
    *x = z;
}
```

А теперь запишем функциональную форму на основе рекурсии:

```
int gcd( int u, int v)
{   if (v == 0) return u;
    else return gcd(v, u % v);
}
```

Как видим, функциональная форма очень лаконична и близка к математическому (рекурсивному) определению функции:

$$\gcd(u,v)=\begin{cases} u & \text{if } v=0; \\ \gcd(v, u \bmod v) & \text{в противном случае.} \end{cases}$$

Отсутствие присваивания проявляется и в другом ограничении: нельзя увидеть внутреннее состояние функции. Значение любой функции зависит только от значений ее аргументов (а также нелокальных переменных). К примеру, значение квадратного корня от определенного числа никогда не меняется, так как значение функции квадратного корня зависит только от значения ее аргумента. Баланс банковского счета изменяется после выполнения вклада, однако его значение зависит не только от размера вклада, но и от старого баланса (внутреннего состояния счета).

Значение любой функции не может зависеть от порядка вычисления ее аргументов. По этой причине функциональное программирование применяется в параллельных приложениях. Свойство, согласно которому значение функции зависит только от значений ее аргументов (и нелокальных переменных), называют *прозрачностью ссылок*. Например, функция `gcd()` прозрачна по ссылкам, так как ее значение определяется лишь значениями ее аргументов. С другой стороны, функция `rand()`, возвращающая псевдослучайное значение, не может быть про-

зрачна по ссылкам, поскольку зависит от состояния автомата (и истории предыдущих вызовов). Интересно, что функция без параметров, прозрачная по ссылкам, должна возвращать все время одно и то же значение; это означает, что она ничем не отличается от константы. Из-за этого функциональный язык вправе считать, что функция без параметров вовсе не является функцией.

Прозрачность ссылок и отсутствие присваиваний существенно упрощают семантику функциональных программ. Здесь отсутствует понятие состояния, поскольку нет понятия области памяти с изменением значения. Среда выполнения ассоциирует имена только со значениями (а не с областями памяти); как только имя вводится в среду, его значение уже никогда не изменится. Семантику функционального программирования иногда называют *семантикой значений*, в противовес семантике сохранения (памяти) и указателей. В самом деле, отсутствие понятия «локальное состояние» противопоставляет функциональное программирование объектно-ориентированному программированию, где вычисления проявляются как изменения локального состояния объектов.

Функции являются главными конструкциями функционального программирования, для которых разрешены самые различные варианты использования, без существенных ограничений. В частности, функции должны рассматриваться и как значения (величины), которые могут вычисляться другими функциями и быть параметрами для других функций. Иными словами, в функциональном программировании функции являются *величинами данных первого класса*.

Например, одной из наиболее распространенных операций над функциями считается композиция. *Композиция* сама по себе является функцией, параметрами которой служат две другие функции. Результат композиции — это функция, значения которой определяются, в свою очередь, результатом применения второй функции-параметра к результату работы первой функции-параметра. Такие функции, которые имеют параметры, являющиеся функциями, или формируют результат в виде функции, или делают и то и другое, называются *функциями высшего порядка*.

Математически операция композиции «o» определяется следующим образом:

```
If f: X → Y and g: Y → Z, then
       g o f: X → Z определяется как (g o f)(x) = g(f(x)).
```

Например, если

```
f(x) ≡ x + 1,
g(x) ≡ 7 * x,
```

тогда `h ≡ f o g` определяется в форме:

`h(x) ≡ g(f(x))`, или `h(x) ≡ (7 * x) + 1.`

Выводы

Перечислим основные особенности функциональных языков программирования:

1. Все процедуры являются функциями с явным отличием входных величин (параметров) от выходных величин (результатов).
2. В чисто функциональном программировании нет присваиваний. После того как переменная привязывается к значению, она ведет себя как константа.

3. В чисто функциональном программировании нет циклов. Вместо циклов используются рекурсивные вызовы.
4. Значение функции зависит только от значений ее параметров и не определяется порядком их вычисления или той цепочкой вычислений, которая привела к вызову.
5. Функции считаются величинами данных первого класса.

Язык Scheme — классический функциональный подход

В конце 50-х и начале 1960-х годов команда из Массачусетского технологического института во главе с Джоном Маккарти (John McCarthy) разработала первый язык, который вобрал в себя многие черты современных функциональных языков [75]. Язык получил название Lisp (LISt Processing), поскольку использовал в качестве базовой структуры список; он основывался на идеях лямбда-исчисления Алонзо Черча. Язык Lisp демонстрировал ряд особенностей, которые, строго говоря, не имели непосредственного отношения к функциональному программированию, но оказали сильное влияние на все последующие функциональные языки. К ним относятся:

1. Единое представление как программ, так и данных с помощью общей структуры данных — списка.
2. Определение языка с помощью интерпретатора, написанного на самом языке — так называемого *метациклического интерпретатора*.
3. Автоматическое управление памятью с помощью системы поддержки периода вычислений.

К сожалению, единый стандарт на язык Lisp отсутствует. В частности, оригинальная версия Lisp и ее преемники не сформировали единого отношения к функциям как величинам данных первого класса. Кроме того, они применяли динамическую область видимости для нелокальных ссылок. Тем не менее, два диалекта Lisp, которые используют статическую область видимости и дают единое толкование функций, приобрели статус стандарта. Первый диалект, Common Lisp, был разработан совместными усилиями в начале 80-х годов. Второй диалект, язык Scheme, был создан в Массачусетском технологическом институте в середине 70-х годов.

Выражения языка Scheme

Все программы и данные в языке Scheme считаются выражениями. Различают два типа выражений: атомы и последовательности из выражений, заключенные в скобки. Атомы подобны литеральным константам и идентификаторам императивного языка, они включают в себя числа, символы, строки, имена, функции и т. д. Выражение в скобках представляет собой просто последовательность из нуля или более выражений, разделенных пробелами и помещенных в круглые скобки. Таким образом, синтаксис схемы достаточно прост, представим его средствами расширенной BNF:

```
<expression> ::= <atom> | '(' {<expression>} ')'
<atom> ::= <number> | <string> | <symbol> | <character> | <Boolean>
```

Здесь первое правило говорит, что выражением может быть либо атом, либо открывающая скобка, за которой следует нуль или более выражений, за которыми следует закрывающая скобка. Второе правило определяет пять разновидностей атомов (в действительности их больше).

Когда выражения в скобках представляют данные, мы будем называть их списками. Примеры корректных выражений для Scheme приведены в табл. 20.1.

Таблица 20.1. Выражения языка Scheme

Выражение	Его смысл
47	Целое значение
"hello"	Строка
#T	Булево значение "true"
#\a	Символьное значение 'a'
(4.1 4.2 5.1)	Список вещественных чисел
a	Символ
hello	Другой символ
(+ 5 7)	Список, состоящий из символа + и двух целых чисел
(* (+ 2 3) (/ 6 2))	Список, состоящий из символа, за которым следуют два списка

Смысл выражения в Scheme определяется правилом вычисления. Стандартное правило вычислений задается следующим образом:

1. Литералы-атомы, такие как числа, символы и строки, представляют себя сами.
2. Символы, отличные от ключевых слов, считаются идентификаторами или переменными, для которых в текущей среде выполняется поиск значений; найденные значения замещают символы. Среду в Scheme образует таблица символов, связывающая идентификаторы со значениями.
3. Для выражения в скобках или списка вычисление производится одним из двух следующих способов:
   - Если первый элемент в выражении является ключевым словом в Scheme, таким как `if` или `let`, то к оставшейся части выражения применяется специальное правило вычисления. Выражения в Scheme, предваряемые ключевыми словами, называются *специальными формами* и используются для управления порядком (потоком) вычислений. Их мы будем обсуждать отдельно.
   - В противном случае выражение в скобках задает применение функции. Каждое выражение в скобках вычисляется рекурсивно, в некотором произвольном порядке; первое выражение должно вычислять функцию. Далее эта функция применяется к остальным величинам, которые считаются ее аргументами.

Применим эти правила к перечисленным выражениям:

- Выражения `42`, `"hello"`, `#T`, и `#\a` представляют сами себя.
- Для выражений `a` и `hello` выполняется поиск в среде, возвращающий их значения.

- Выражение (+ 5 7) вычисляется поиском в среде значения для + — возвращаемым значением становится предопределенная функция сложения, а затем функция сложения применяется к значениям величин 5 и 7, которые являются числами 5 и 7 (поскольку константы представляют сами себя). Следовательно, возвращается значение 12.
- Аналогичным образом вычисляется (* (+ 2 3) (/ 6 2)), возвращающее значение 15.
- Выражение (4.1 4.2 5.1) вычислить нельзя, поскольку его первое выражение 4.1 является константой, а не функцией. Это выражение не является ни специальной формой, ни применением функции и приводит к ошибке, если будет предпринята попытка вычисления.

Правило вычисления в Scheme подразумевает, что все выражения должны записываться в префиксной форме. Кроме того, подразумевается, что значение функции (как объекта) легко отличить от вызова функции:

- значение функции представляется первым выражением в применении функции;
- вызов функции заключен в круглые скобки.

Следовательно, диалог с интерпретатором языка Scheme примет следующий вид:

```
> +
#[PROCEDURE: +]
> (+ 5 7)
12              ; вызов процедуры + с аргументами 5 и 7
```

Отметим, что в языке Scheme построчные комментарии начинаются с символа точка с запятой. Сопоставление выражений в языках C и Scheme иллюстрирует табл. 20.2.

Таблица 20.2. Выражения в языках C и Scheme

Язык C	Язык Scheme
3 + 4 * 5	(+ 3 (* 4 5))
(a == b) && (a != 0)	(and (= a b) (not (= a 0)))
gcd(10,35)	(gcd 10 35)
gcd	gcd
getchar()	(read-char)

Правило вычисления в языке Scheme определяет конкретный порядок вычисления. Все подвыражения вычисляются в первую очередь, следовательно, синтаксическое дерево выражения проходится от листьев к корню. Например, выражение в Scheme (* (+ 2 3) (+ 4 5)) вычисляется в следующем порядке: сначала выполняются оба сложения, а затем вычисляется результирующее выражение (* 5 9).

Это соответствует вычислению выражения (2 + 3) * (4 + 5) в языке C, где выражения записываются в инфиксной форме.

Специальные формы

Так как данные и программы в языке Scheme используют одинаковый синтаксис, при представлении данных непосредственно в программах могут возникнуть

проблемы. Как уже говорилось, список чисел представляется их записью внутри круглых скобок, где числа отделяются друг от друга пробелами:

```
(2.1 2.2 3.1)
```

Если передать этот список прямо интерпретатору, будет сгенерирована ошибка, так как Scheme попытается вычислить его как вызов функции. Следовательно, возникает необходимость предотвратить вычисление списка, посчитав его списковым литералом (подобно простому атомарному литералу, который сам себя представляет). Это достигается применением специальной формы, предотвращающей вычисление. Данная форма задействует ключевое слово `quote`, причем следующее за ним выражение возвращается без вычисления. Приведем пример диалога с интерпретатором:

```
> (2.1 2.2 3.1)
Error: the object 2.1 is not a procedure
> (quote (2.1 2.2 3.1))
(2.1 2.2 3.1)
> (quote scheme)
scheme
```

Последнее выражение возвращает символ. В языке Scheme символы соответствуют отдельному типу данных. Символы рассматриваются как идентификаторы или как имена, которые ссылаются на значения. После слова `quote` символ интерпретируется как литерал. Символы могут добавляться к спискам или сравниваться с другими символами (проверка эквивалентности). Включение символов как явного типа данных превращает язык Scheme в удобное средство для обработки любой символьной информации.

Специальная форма `quote` очень популярна, поэтому для нее предусмотрено специальное сокращение — апостроф:

```
> '(2.1 2.2 3.1)
(2.1 2.2 3.1)
```

Поскольку все конструкции в Scheme являются выражениями, нужны выражения для управления выполнением. Циклы моделируются рекурсивным вызовом, а для моделирования ветвления предусмотрены специальные формы:

- форма `if`, которая подобна конструкции if-else;
- форма `cond`, похожая на конструкцию if-elsif (`cond` обозначает условное выражение).

Примеры условных форм:

```
(if (= a 0) 0         ; if a = 0 then return 0
        (/ 1 a))         ; else return 1/a
    (cond((= a 0) 0)         ; if a = 0 then return 0
        ((= a 1) 1)         ; elsif a = 1 then return 1
    (else (/ 1 a)))         ; else return 1/a
```

Семантика выражения `(if exp1 exp2 exp3)` подсказывает, что `exp1` вычисляется первым; если значением `exp1` является булева величина `false (#f)`, тогда вычисляется `exp3` и его значение возвращается выражением `if`; в противном случае вычисляется `exp2` и возвращается его значение. Кроме того, выражение `exp3` может отсутствовать; в этом случае при значении `false` для `exp1` значение всего выражения не определено.

Аналогично, семантика (cond exp1 . . . expn) подсказывает, что каждое expi должно быть парой выражений: expi = (fst snd). Каждое выражение expi рассматривается по порядку, и вычисляется его первая часть. Если оказалось, что значение fst равно #T (true), тогда вычисляется snd и его значение возвращается из выражения cond. Если вычисление ни одного из условий не привело к значению #T, тогда вычисляется выражение из ветви else и возвращается как значение выражения cond (считается, что вычисление самого ключевого слова else всегда приводит к значению #T). Если же ветви else нет и ни одно из условий не вычислило #T, тогда результат выражения cond не определен.

Заметим, что формы if и cond не подчиняются стандартному правилу вычисления выражений. В противном случае все их аргументы вычислялись бы каждый раз (независимо от их значений) и формы становились бы совершенно бесполезны в качестве механизмов управления. Вместо этого вычисления аргументов в таких специальных формах задерживаются до подходящего момента. Применения функций в Scheme ориентированы на передачу аргументов по значению, а специальные формы в Scheme и Lisp используют отложенные вычисления.

Другой важной специальной формой является let, задающая связывание имен переменных со значениями внутри выражения:

```
> (let ((a 2) (b 3)) (+ a b))
5
```

Первым выражением в форме let является список связывания, который ассоциирует имена со значениями. В нашем примере список задает два связывания: a придается значение 2, а b — значение 3. Далее имена с приданными значениями используются во втором выражении, которое считается телом формы let и чье значение является возвращаемым значением let. Форма let обеспечивает локальную среду и область видимости для установки имен переменных точно так же, как и объявления временных переменных в языках с блочной структурой (Java, Python). Значения этих переменных доступны внутри формы let, но не вне ее.

Создание функций

Для разработки программ язык Scheme предусматривает способ создания функций. Для этого применяется специальная форма lambda (термин lambda заимствован из лямбда-исчисления Черча). Лямбда-выражение имеет следующую форму:

```
(lambda param-list body)
```

Когда Scheme вычисляет форму lambda, он создает функцию с определенными формальными параметрами и тело программного кода, обеспечивающее вычисления при применении функции. Например, функция для вычисления площади круга имеет вид:

```
> (lambda (radius) (* 3.14 (* radius radius)))
#<procedure>
```

Чтобы задать применение этой функции к аргументу, функцию и аргумент помещают во внешнюю пару скобок:

```
> ((lambda (radius) (* 3.14 (* radius radius))) 10)
314.0
```

Поскольку к функциям удобно обращаться по именам, можно связать имя (например, `acircle`) с лямбда-выражением в форме `let` и применить это имя к аргументу в теле `let`:

```
> (let ((acircle (lambda (radius) (* 3.14 (* radius radius)))))
    (acircle 100))
3140.0
```

К сожалению, `let` нельзя использовать для определения рекурсивных функций, поскольку в связываниях `let` нельзя обращаться к самому себе или к другому коду. Поэтому следующий код будет генерировать ошибку:

```
(let ((factorial (lambda (n) (if (= n 0) 1 (* n (factorial (- n 1)))))))
     (factorial 10))
```

Для подобных ситуаций существует другая специальная форма, именуемая `letrec`, которая работает точно так же и обеспечивает произвольные рекурсивные ссылки внутри списка связывания:

```
(letrec ((factorial (lambda (n) (if (= n 0) 1 (* n (factorial (- n 1)))))))
     (factorial 10))
```

Формы `let` и `letrec` вводят переменные, которые видимы внутри их области видимости и существуют в период жизни этих форм. Для обеспечения нового, глобального связывания переменной на верхнем уровне среды Scheme можно использовать специальную форму `define`:

```
> (define acircle (lambda (radius) (* 3.14 (* radius radius))))
> (define x 5)
> (acircle 10)
314.0
> (acircle x)
78.5
>
```

Динамическая проверка типа

Семантика языка Scheme определяет динамическую или скрытую проверку типа. Считается, что только значения, а не переменные имеют типы. Типы значений не проверяются до тех пор, пока в этом не возникнет острая необходимость, причем в период вычислений. Автоматическая проверка происходит лишь перед применением примитивной функции, такой как, например, +, к ее аргументам. Если один из аргументов не является числом, среда Scheme формирует сообщение об ошибке и останавливает вычисления. Напротив, в определенных программистом функциях типы аргументов автоматически не проверяются. Например, следующий фрагмент, где определяется и применяется функция `square`,

```
> (define square (lambda (n) (* n n)))
> (square "hello")
```

приводит к потенциальной ошибке типа для функции:

```
. . *: expects type <number> as 1st argument, given: "hello"
```

Ошибка сгенерирована автоматически, но, возможно, не в том месте, где это было бы действительно полезно. Программист может проверить тип любого значения встраиванием в условные выражения функций распознавания, таких как `number?`

и `procedure?`. Тем не менее необходимость писать код для проверки типов снижает скорость создания и выполнения программ.

Хвостовая и не хвостовая рекурсия

В императивных языках программирования циклы всегда предпочтительнее рекурсии в силу значительных накладных затрат на вызов процедуры. Однако функциональный стиль программирования поддерживает именно рекурсию. Возникает вопрос: как минимизировать неэффективность рекурсии? В языках, подобных Scheme, предлагается делать рекурсивными последние шаги вычисления любой функции. Другими словами, рекомендуется создавать функции с хвостовой рекурсией. Компилятор языка Scheme транслирует функции с хвостовой рекурсией в код, выполняемый в виде цикла, без дополнительных накладных затрат на вызов функции, за исключением верхнего уровня вызова. Рассмотрим не хвостовое рекурсивное определение функции факториала:

```
> (define factorial
    (lambda (n)
        (if (= n 1)
           1
            (* n (factorial (- n 1))))))
> (factorial 6)
720
```

В этой версии значение, возвращаемое вызовом функции, должно умножаться на `n`, которое являлось аргументом предыдущего вызова. Следовательно, стек периода вычислений должен отслеживать значение этого аргумента по мере «раскручивания» рекурсии. Эта версия факториала приводит, таким образом, к линейному росту задействованной памяти, а также к значительному падению производительности по сравнению с не рекурсивной версией, использующей цикл.

А теперь приведем определение функции факториала с хвостовой рекурсией:

```
> (define factorial
    (lambda (n result)
        (if (= n 1)
           result
            (factorial (- n 1) (* n result)))))
> (factorial 6 1)
720
```

При хвостовой рекурсии вся работа по вычислению значений выполняется в те моменты, когда аргументы вычисляются перед каждым рекурсивным вызовом. Как и раньше, значение `n` уменьшается. Второй аргумент, `result`, используется для накапливания промежуточных значений на нисходящем пути через рекурсивные вызовы. На верхнем уровне вызова значением `result` является 1, а далее значение умножается на текущее значение `n` перед каждым рекурсивным вызовом. Поскольку после каждого рекурсивного вызова не остается никакой работы, стеку периода вычислений не нужно запоминать аргументы предыдущих вызовов. Таким образом, компилятор Scheme может транслировать функцию факториала с хвостовой рекурсией в некоторый код, который использует цикл, содержащий специальные формы `begin`, `while` и `set!`:

```
> (define factorial
    (lambda (n result)
    (begin
        (while (> n 1)
            (set! result (result * n))
            (set! n (- n 1)))
       result)))
```

Специальная форма `begin` устанавливает естественный порядок выполнения выражений (слева-направо) и возвращает значение последнего выражения. Специальная форма `set!` устанавливает значение переменной. Специальная форма `while` в языке Scheme отсутствует, но может быть определена как макроподстановка. Конечно, все это похоже на императивный код, который добросовестный «функциональный» программист писать никогда не станет. Однако подобное компромиссное решение позволяет сочетать рекурсивный, функциональный стиль с высокой производительностью (при наличии компилятора для генерации кода, оптимизирующего хвостовую рекурсию).

Структуры данных в языке Scheme

В языке Scheme базовой структурой данных является список. Хотя Scheme поддерживает и другие составные типы данных (например, одномерные массивы и строки), список можно использовать для представления последовательности, записи или любой другой структуры. Например, следующий список описывает бинарное дерево поиска:

```
("man" ("girl" () ("liza" () ()))
    ("boy" ("tom" () ()) () ))
```

Вершина в этом дереве соответствует списку из трех элементов (`name left right`), где `name` является строкой, а `left` и `right` — это дочерние деревья, описывающие листья. Этот список представляет дерево, изображенное на рис. 20.1.

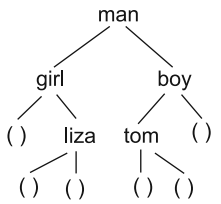


Рис. 20.1. Дерево двоичного поиска, содержащее строковые данные

Язык Scheme предлагает большой набор предопределенных функций для работы со списками, причем базовые функции являются общими для всего семейства языков Lisp — это функции выбора `car` и `cdr`, обеспечивающие доступ к голове и хвосту списка, а также функция-конструктор `cons`, которая добавляет новую голову к существующему списку. Таким образом, если `L` это список `(1 2 3)`, тогда:

```
> (car L)
1
> (cdr L)
(2 3)
```

```
> (cons 4 L)
(4 1 2 3)
```

Имена car и cdr имеют исторические корни. Первым компьютером, на котором был реализован Lisp, стала вычислительная машина IBM 704. В этой виртуальной машине в качестве адреса головы списка использовалось содержимое адресного регистра (Contents of the Address Register, или сокращенно car), а на хвост списка указывало содержимое декрементного регистра (Contents of the Decrement Register — сокращенно cdr).

Все базовые операции для обработки любых списков можно записать как функции, составленные из примитивов car, cdr, cons, и null?. Последняя функция является предикатом, возвращающим true, если список пуст, или false в противном случае. Например, операция append, возвращающая «добавленный список», состоящий из двух списков, может быть записана следующим образом:

```
(define append (lambda (L M)
     (if (null? L) M
         (cons (car L) (append (cdr L) M)))))
```

Операция reverse принимает вид:

```
(define reverse (lambda (L)
     (if (null? L) '()
         (append (reverse (cdr L)) (list (car L))))))
```

В функции reverse использована примитивная функция list, которая составляет список из одного или нескольких элементов:

```
> (list 2 3 4)
  (2 3 4)
> (list '(a b) '(c d))
  ((a b) (c d))
```

В функции reverse вместо (list (car L)) можно применить (cons (car L) '()), так как выражение (list a) эквивалентно выражению (cons a '()).

Положим, что имеется бинарное дерево поиска, организованное в виде списка со следующей структурой:

```
(data l-child r-child)
```

Определим доступ к этому дереву с помощью car и cdr:

```
(define (l-child B) (car ( cdr B)))
(define (r-child B) (car (cdr (cdr B))))
(define (data B) (car B))
```

Теперь мы можем написать процедуру обхода дерева put-tree, которая выводит на экран данные дерева:

```
(define put-tree (lambda (B)
     (cond ((null? B) '() )
         (else (put-tree (l-child B))
             (display (data B))
             (newline)
             (put-tree (r-child B))))))
```

Программирование в языке Scheme

Программирование в языке Scheme, как и в любом другом функциональном языке, основано на рекурсии для выполнения циклов и других повторяющихся операций.

Стандартным приемом для применения к списку повторяющихся операций является методика «cdr вниз и cons вверх», означающая рекурсивное применение операции к хвосту списка, с последующим конструированием нового списка из текущего результата посредством операции cons. Примером может служить процедура append, приведенная в предыдущем подразделе. Другим примером может быть процедура возведения в квадрат всех элементов в некотором списке:

```
(define square-list (lambda (L)
     (if (null? L) '()
         (cons (* (car L) (car L)) (square-list (cdr L))))))
```

Еще одним примером циклических вычислений служит процедура, выводящая квадраты целых чисел в диапазоне от некоторой нижней до определенной верхней границы:

```
(define put-squares (lambda (low high)
     (cond ((> low high) '())
     (else (display (* low low))
         (newline)
         (put-squares (+ 1 low) high))))
```

Вызов (put-squares 1 500) приведет к генерации требуемого результата.

В примере put-squares для управления рекурсией использовался дополнительный параметр low, играющий роль переменной цикла. Ранее мы отмечали, что в подобных ситуациях желательны процедуры с хвостовой рекурсией, которые легко превращаются трансляторами в оптимизирующие циклы. По сути, процедура put-squares реализует хвостовую рекурсию.

Мы уже демонстрировали методику использования параметра-аккумулятора для превращения не хвостовой рекурсии в хвостовую. Применим эту методику к функции square-list, разработав процедуру square-list-new с дополнительным параметром для накапливания промежуточного результата:

```
(define square-list-new (lambda (L list-so-far)
     (if (null? L) list-so-far
     (square-list-new (cdr L)
         (append list-so-far (list (* (car L) (car L)))))))
```

Теперь мы можем определить square-list как вызов square-list-new с () в качестве первой аккумулирующей величины:

```
(define square-list (lambda (L) (square-list-new L '())))
```

Аналогичным образом определим reverse-new для reverse и используем новое определение:

```
(define reverse-new (lambda (L list-so-far)
     (if (null? L) list-so-far
         (reverse-new (cdr L) (cons (car L) list-so-far))))
(define reverse (lambda (L) (reverse-new L '())))
```

Функции высшего порядка

В языке Scheme функции являются величинами первого класса, поэтому можно создавать функции, которые принимают другие функции как параметры, а также

функции, которые возвращаются функции как значения. Такие функции называют функциями высшего порядка.

Создадим функцию `map` с параметром-функцией, которая применяет эту другую функцию ко всем элементам списка. Используем функцию `square` как параметр для функции `map`, а саму настроенную функцию `map` передадим как параметр в функцию `square-list`, обеспечивающую получение списка квадратов:

```
(define map (lambda (f L)
     (if (null? L) '()
         (cons (f (car L)) (map f (cdr L))))))
(define square (lambda (x) (* x x))
(define square-list (lambda (L) (map square L))
```

Приведем пример функции, которая принимает функцию-параметр `f` и возвращает функцию (назовем ее `fdouble`) как значение:

```
(define make-double (lambda (f)
     (lambda (x) (f x x)))
```

Функция `make-double` предполагает, что `f` является функцией с двумя параметрами, и создает функцию, которая при вызове `f` повторяет параметр `x` (передает параметр `x` два раза). Функция-значение (`fdouble`), возвращаемое функцией `make-double`, создается локальным определением, которое записывается в конце `make-double` и поэтому считается ее возвращаемым значением. Отметим, что `x` не является параметром функции `make-double`, а является параметром создаваемой ею функции.

Функцию `make-double` можно использовать для получения функции возведения в квадрат `square` и функции удвоения `double` (функции, которая удваивает числовое значение):

```
(define square (make-double *))
(define double (make-double +))
```

Символы «*» и «+» являются именами функций умножения и сложения. Их определения (функциональные значения) передаются как параметры в процедуру `make-double`, которая возвращает функциональные значения, назначаемые именам `square` и `double`. Заметим, что здесь используется простая форма `define`, которая назначает вычисленные значения, а не определяет функцию с заданными телом. Достаточно написать

```
(define n 7)
```

чтобы назначить значение `7` имени `n`. Выражение

```
(define square (make-double *))
```

назначает функциональное значение, возвращаемое функцией `make-double` имени `square`.

Данное свойство функциональных языков — возвращать функциональные значения из функций высшего порядка — означает, что среда поддержки вычислений в функциональных языках сложнее, чем стандартная, основанная на стеке, среда императивного языка. Возвращение памяти, используемой функциями, требует развитой методики автоматического управления памятью, например сбора мусора.

Статическая область видимости

Ранние диалекты языка Lisp характеризовались динамической областью видимости, в то время как современные диалекты (Scheme и Common Lisp) приняли на вооружение статическую область видимости. Термин статическая область видимости относится к области программного текста, в пределах которого видимо объявление переменной. Это значит, что при статической области видимости смысл или значение данной переменной определяется простым чтением текста программы. Напротив, в языках с динамической областью видимости смысл переменной зависит от контекста периода вычислений.

Концепция области видимости очень важна для языков с блочной структурой, где возможны вложения объявлений переменных. К примеру, рассмотрим фрагмент кода на Scheme, где в форму `let` вложена другая форма `let`:

```
> (let ((a 2) (b 3))
     (let ((a (+ a b)))
     (+ a b)))
8
```

Правило области видимости утверждает, что область видимости переменной распространяется до конца блока, где она объявлена, включая любые вложенные блоки, за исключением тех вложенных блоков, где переменная переопределена. Следовательно, в этом примере значение первого экземпляра `(+ a b)` равно `5`, а значение второго экземпляра `(+ a b)` равно `8`.

Правило области видимости существенно усложняется при включении функций со свободными переменными. *Свободная переменная* — это переменная, на которую ссылаются в функции, но которая не является формальным параметром этой функции и не ограничена вложенной функцией. Ограниченная переменная — это переменная внутри функции, являющаяся ее формальным параметром. Приведем пример определения и использования функции, которая содержит свободную переменную, а также ограниченную переменную:

```
> (let ((pi 3.14))
    (let ((acircle (lambda (radius) (* pi (* radius radius)))))
    (let ((pi 3.1416)) (acircle 10))))
314.0
```

Функция `acircle` содержит одну ограниченную переменную, `radius`, и одну свободную переменную, `pi`. Ограниченная переменная `radius` является также формальным параметром функции и принимает свое значение в контексте вызова функции, который ограничивает ее значением `10`. Значение свободной переменной `pi` должно быть равно `3.14` при ссылке на нее из применения функции. Следовательно, свободная переменная `pi` устанавливается во внешнем `let` и сохраняет свое значение как во внешнем `let`, который является контекстом определения функции, так и во внутреннем `let`, который является контекстом вызова функции. Напротив, если бы в языке Scheme была принята динамическая область видимости (подобно старым версиям Lisp), тогда значение `pi`, используемое при применении функции, должно быть взято из внутреннего `let`, то есть из контекста вызова функции. Это значит, что приложение выработало бы значение `314.16`. В этом случае свободная переменная функции получила бы ту же семантику, что

и ее формальный параметр. Следовательно, при динамической области видимости значения свободных переменных и формальных параметров меняются в зависимости от контекста вызова функции.

Статическая область видимости, поскольку она фиксирует смысл свободных переменных в некотором месте программного кода, упрощает чтение и проверку программ (по сравнению с динамической областью видимости).

Настройка — специализация языка Scheme

Напомним, что программы и данные на языке Scheme имеют одинаковый синтаксис. К примеру, с точки зрения данных, выражение `(f x y)` — это список из трех символов, которые можно обработать с помощью операций над списками. С точки зрения программы, выражение `(f x y)` — это применение функции с именем `f` к значениям ее аргументов, именуемым переменными `x` и `y`. Возможность создавать, обрабатывать и преобразовывать списки символов, а затем вычислять их как программы придает языку Scheme особые черты, позволяет применять его для металингвистической обработки: обработки отношений, производства логических выводов и т. д.

Настройка на подобные предметные области требует решения трех задач: создания синтаксиса, семантики новых конструкций, а также реализации нового интерпретатора.

Для иллюстрации этой идеи обсудим возможность моделирования формы `let` иными средствами языка. Напомним, что `let` позволяет интерпретатору вычислять выражение в новой среде, создаваемой связыванием набора временных переменных с их значениями. Приведем пример формы `let`:

```
(let ((a 5) (b 7))
    (* a b))
```

Но ведь то же самое действие можно задать с помощью применения лямбда-формы к ее аргументам:

```
((lambda (a b) (* a b)) 5 7)
```

Видим, что временные переменные формы `let` становятся формальными параметрами `lambda`, а их начальные значения превращаются в аргументы применения `lambda`. Соответственно, тело формы `let` стало телом `lambda`.

Процесс интерпретации формы `let` включает два шага. Во-первых, форма `let` переводится в эквивалентное применение лямбда-выражения. Во-вторых, вычисляется новое представление. Если существует некоторая функция (например, `let-to-app`), которая выполняет требуемое синтаксическое преобразование, тогда мы можем применить функцию `eval` из языка Scheme для проверочного вычисления:

```
> (eval (let-to-app '(let ((a 5) (b 7))(* a b))))
35
```

Аргумент функции `let-to-app` (выражение `let`) предваряется апострофом для того, чтобы трактовать его как данные (список элементов). Функция `eval` обеспечивает вычисление результата функции `let-to-app`, которая список элементов превращает в применение лямбда-выражения.

Осталось определить функцию `let-to-app`, которая преобразует любое выражение `let` в соответствующее применение лямбда-выражения. В реализации задействуем различные функции для обработки списков и символ апостроф (сокращенную форму `quote`). Комментарии описывают содержание шага действий:

```
(define let-to-app (lambda (let-form)
     (append (list                        ; начало приложения
     (list 'lambda                        ; начало лямбда-выражения
     (map car (cadr let-form))            ; формальные параметры
     (caddr let-form)))                   ; тело лямбда-выражения
     (map cadr let-form)))))              ; аргументы выражения
```

Кроме того, с помощью макроопределений можно определить синтаксические формы для нового языка (если их еще нет). Тогда программисту при использовании новых форм не придется прибегать к цитированию и явному вычислению с помощью `eval`. После подключения новых форм компилятор Scheme будет автоматически выполнять требуемые синтаксические преобразования и генерировать код, вычисляемый в период выполнения.

Язык ML — функциональный подход со статической типизацией

Язык ML (MetaLanguage) является функциональным языком, который значительно отличается от диалектов Lisp, в том числе от языка Scheme. Во-первых, синтаксис ML более похож на синтаксис языка Algol и не требует использования большого количества скобок. Во-вторых, он считается статически типизированным языком, что означает: тип каждого выражения определяется перед выполнением, а значит, можно проверять корректность и согласованность типов. Конечно, обычному программисту на языке Lisp не по душе ограничения статической системы типизации, но система статической типизации в ML обеспечивает ряд важных преимуществ. С одной стороны, она повышает безопасность языка, поскольку ошибки отыскиваются перед выполнением программы. Это способствует высокому качеству программных приложений и особенно важно в учебных целях. Кроме того, статическая система типизации ML улучшает эффективность, поскольку предопределяет размер потребной памяти и позволяет отказаться от проверки типов в период вычислений. Статическая система типизации ML характеризуется чрезвычайной гибкостью. Она не настаивает, подобно языкам C или Ada, на объявлении программистом всех типов и предоставляет возможности параметрического полиморфизма, когда для переменной можно задать один из некоторого набора типов. Первая версия языка ML была разработана в конце 70-х как часть Эдинбургской системы доказательства правильности программ, созданной командой под руководством Робина Милнера. Милнер предложил эффективную систему логического вывода типа, основанную на сопоставлении паттернов. Эта форма проверки типов является ключевой характеристикой языков ML и Haskell.

Мы будем рассматривать базис современной стандартной версии языка ML, сравнивая его с языком Scheme.

Функции и величины языка ML

В языке ML базовой программой, как и в языке Scheme, является объявление функции:

```
> fun fact n = if n = 0 then 1 else n * fact(n - 1);
val fact = fn: int -> int
```

Зарезервированное слово `fun` в ML вводит объявление функции. Идентификатор, следующий за `fun`, считается именем функции; за именем функции перечисляются параметры. Перечень параметров замыкается знаком «равно». После знака «равно» записывается тело функции.

Система ML предоставляет объявление возвращаемой величины и типа этой величины. В данном примере функцией является `fact`, ее величиной считается `fn` с типом `int -> int`, который означает, что функция `fact` отображает из *области определения* целых чисел в *область значений* целых чисел. Действительно, можно задать объявление типа функции `fact` и типа ее параметра в виде:

```
> fun fact (n: int): int = if n = 0 then 1
else n * fact (n - 1);
val fact = fn: int -> int
```

После объявления функция может быть вызвана:

```
> fact 7;
val it = 5040 : int
```

Среда ML предоставляет возвращаемое значение и его тип (`it` — это имя вычисляемого выражения). Правило вычислений в ML, по сути, такое же, как и в Scheme: `fact` задает вычисление функции, которое применяется к `7`, тип аргумента должен совпадать с типом параметра функции. Функция вызывается, ее возвращаемое значение печатается вместе с типом. В ML не нужна функция `quote` (как в языке Scheme) для предотвращения вычислений, поскольку данные отличны от программ. В ML круглые скобки почти не употребляются, так как система способна распознать смысл элементов, основываясь на их позиции.

Величины в ML можно определять с помощью ключевого слова `val`:

```
> val Pi = 3.14159;
val Pi = 3.14159 : real
```

Арифметические операции в ML записываются в инфиксной форме, так же, как в C или Ada. Этим язык отличается от унифицированной префиксной нотации для семейства Lisp. В итоге в ML появляются проблемы ассоциативности и приоритетности.

Конечно, язык ML придерживается стандартных математических соглашений по поводу стандартных математических операций. К примеру, `2 + 3 * 4` означает `2 + (3 * 4)`. В ML с помощью ключевого слова `op` можно задать префиксную форму записи, поэтому `2 + 3 * 4` можно записать так:

```
> op + (2 , op * ( 3,4));
val it = 14 : int
```

Бинарные арифметические операции в качестве аргументов принимают пары целых чисел. Типы этих пар являются элементами декартового произведения типов или кортежем `int * int`:

```
> (2,3);
val it = (2,3) : int * int
> op +;
val it = fn : int * int -> int
```

Списки в языке ML

Списки в ML также играют важную роль, как и в любом функциональном языке. Однако они не настолько универсальны, как в Lisp, поскольку в ML программы не являются списками. Списки в ML указываются квадратными скобками, элементы в них отделяются запятыми. К примеру, список целых 1, 2 и 3 представляется как [1, 2, 3]:

```
> [1,2,3];
val it = [1,2,3] : int list
```

Язык ML определяет тип этого списка как `int`. Строгая типизация в ML диктует, что все элементы списка должны иметь одинаковый тип. Рассмотрим пример структуры, нарушающей это соглашение и смешивающей типы элементов в списке:

```
> [1,2,3.1];
Error: operator and operand don't agree [literal]
    operator domain: int * int list
    operand: int * real list
    in expression:
        2 :: 3.1 :: nil
```

При необходимости смешать данные разных типов следует использовать кортеж:

```
> (1,2,3.1);
val it = (1,2,3.1) : int * int * real
```

Сообщение об ошибке, сгенерированное в предыдущем примере, подсказывает нам правила конструирования списков в ML, которые похожи на правила конструирования списков в Scheme. Здесь операция `::` соответствует операции `cons` в Scheme и создает список из элемента, стоящего слева от знака операции (который становится головой списка), и предыдущего уже созданного списка, стоящего справа от значка операции (который становится хвостом списка). К примеру:

```
> op :: ;
    val it = fn : 'a * 'a list -> 'a list
> 2 :: [3,4];
val it = [2,3,4] : int list
```

Обратите внимание, что запись операции `::` содержит запись переменной `'a`, поскольку операция `::` должна применяться к списку, содержащему величины любого типа. Таким образом, `::` — это функция, которая из элемента любого типа `'a` и любого списка из элементов того же типа `'a` конструирует список такого же типа `'a`.

В языке ML каждый список конструируется последовательностью применения операций `::`, как показано в следующем фрагменте:

```
> 1 :: 2 :: 3 :: [];
val it = [1,2,3] : int list
```

Здесь `[]` обозначает пустой список, который также может записываться посредством ключевого слова `nil`.

Язык ML содержит операции, соответствующие `car` и `cdr` из Scheme, правда, называются они здесь `hd` и `tl` (для головы `head` и хвоста `tail`). К примеру:

```
> hd [1,2,3];
val it = 1 : int
> tl [1,2,3];
val it = [2,3] : int list
```

В ML эти функции применяют редко, в отличие от их аналогов в Scheme, поскольку ML предлагает более удобный механизм сопоставления с паттерном. В ML для выделения головы и хвоста списка `L` достаточно записать для списка `h::t`. Среда языка ML выделит голову как `h`, а хвост как `t` (в соответствии с сопоставлением паттерна для списка `L`). Если `L` пуст, паттерн не сопоставится и будет сгенерировано подобное предостережение:

```
> val h::t = [1,2,3];
Warning: binding not exhaustive
    h :: t = ...
val h = 1 : int
val t = [2,3] : int list
```

Обычно сопоставление с паттерном используется в определениях функций и выражениях выбора. Например, функцию `append` для списка можно записать в следующей форме:

```
> fun append ([] , L) = L
    | append (h :: t , L) = h :: append (t , L);
val append = fn : 'a list * 'a list -> 'a list
```

Это эквивалентно следующему определению, использующему выражение `case`:

```
> fun append (x,y) =
    case x of
         [] => y |
         (h::t) => h :: append (t,y);
```

Сопоставление с паттерном может использоваться практически во всех определениях функций, где требуется анализ выбора; оно может устранить большинство использований условных выражений. К примеру, рекурсивная функция факториала может быть переписана на основе сопоставления с паттерном следующим образом:

```
fun fact 0 = 1 | fact n = n * fact (n - 1);
```

Здесь второй паттерн — переменная `n` — не сопоставляется ни с чем, но его позиция как второго паттерна неявно свидетельствует, что он не должен быть равен нулю; если же имеет место обратный факт, то должен сопоставляться первый паттерн (целый литерал `0`).

Паттерны в ML могут использовать символы подстановки, записываемые как знаки подчеркивания. Символы подстановки являются паттернами, которые сопоставляются с чем угодно, независимо от содержания. К примеру, можно определить собственную версию функции `hd`:

```
fun hd (h::_) = h;
```

На самом деле, это определение инициирует предупреждение («сопоставление неполное»), поскольку здесь не определены действия `hd` в случае пустого списка. В лучшем определении предусмотрим и этот вариант:

```
fun hd (h::_) = h | hd [] = raise Empty;
```

Проверка типов в языке ML

При строгой типизации язык ML остается безопасным и гибким, не навязывающим ограничений на те характеристики, которые считаются приемлемыми для программы. Типичный пример — функция `square`, которую мы определим для вещественных величин:

```
> fun square x: real = x * x;
val square = fn : real -> real
```

При попытке вызова этой функции для целого числа будет генерироваться ошибка, поскольку ML (подобно языку Ada) не будет автоматически конвертировать целое в вещественное:

```
> square 2;
Error: operator and operand don't agree [literal]
    operator domain: real
    operand: int
    in expression:
        square 2
```

Вместо этого нам потребуется вручную конвертировать целое в вещественное с помощью функции преобразования. В следующем примере явно конвертируется тип данных аргумента:

```
> square (real 2);
val it = 4.0 : real
```

Можно определить подобную функцию с тем же именем для целых:

```
> fun square x: int = x * x;
val square = fn : int -> int
```

но язык ML не разрешает перегрузку. Следовательно, это определение просто заменит предыдущее. Можно попытаться определить родовую версию функции `square`, не указывая тип параметра:

```
> fun square x = x * x;
val square = fn : int -> int
```

Тем не менее в этом случае ML просто предположит, что мы посчитали `x` целым числом. Проблема в том, что хотя встроенные арифметические операции и являются перегружаемыми для типов `int` и `real`, перегружаемые пользовательские функции не поддерживаются.

Смежная проблема касается того, что определенные величины в ML не содержат достаточной информации для проверки типа выражения перед его вычислением. Возьмем пустой список `[]`, который имеет тип `'a list`. Если ML вычисляет полиморфную функцию над пустым списком, среда отобразит, что отсутствует достаточная информация о типе `[]`:

```
> rev [];
    type vars not generalized because of value
restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
```

Здесь `rev` — встроенная функция `reverse`, которая реверсирует список и имеет тип `'a list -> 'a list`.

Когда функция `rev` применяется к пустому списку, ML хочет заменить тип переменной `'a` конкретным типом и не может сделать это из-за недостатка информации. Тем не менее ML компенсирует этот недостаток применением типа-пустышки и получает правильный ответ, несмотря на предупреждение. Подстановка конкретного типа позволяет избежать подобной проблемы:

```
- rev []:int list;
val it = [] : int list
```

Еще один специальный случай проверки типов связан с четким разграничением типов, значения которых могут проверяться на эквивалентность, и всех остальных типов. Например, вещественные числа не могут проверяться на равенство:

```
> 2.1 = 2.1;
Error: operator and operand don't agree [equality type required]
    operator domain: ''Z * ''Z
    operand: real * real
    in expression:
        2.1 = 2.1
```

Когда определение полиморфной функции использует операцию эквивалентности, переменные с типами, поддерживающими эту операцию, помечаются двумя апострофами:

```
> fun constList [] = true
    | constList [a] = true
    | constList (a::b::L) = a = b andalso constList (b::L);
val constList = fn : ''a list -> bool
```

В самом деле, операция эквивалентности сама по себе может применяться только к величинам типов, поддерживающим эту операцию:

```
> op =;
val it = fn : ''a * ''a -> bool
```

Ввод и вывод в языке ML

В языке ML имеется библиотечный пакет, который содержит предопределенные стандартные ресурсы для выполнения ввода и вывода. Основные функции ввода и вывода находятся в структуре `TextIO`, простейшими из них являются функции `inputLine` и `output`:

```
> open TextIO; (* разыменование структуры TextIO *)
... (* интерпретатор ML перечисляет содержимое TextIO *)
> output(stdOut,inputLine(stdIn));
(* читать и писать строку текста *)
Hello, world!
Hello, world!
val it = () : unit
>
```

Отметим, что возвращаемой величиной операции `output` является величина `()`, которая имеет тип `unit`. Этот тип подобен типу `void` в языке C, но в ML каждая функция должна возвращать некоторую величину, поэтому тип `unit` имеет точно одно значение `()`, которое означает «нет фактического значения».

Достаточно популярны функции для чтения и записи строк. Для преобразований между строками и числами используются функции `toString` и `fromString` из структур `utility` для соответствующих типов (например, из структуры `Int` для целых типов):

```
> Int.toString;
val it = fn : int -> string
> Int.fromString;
val it = fn : string -> int option
```

Здесь возникает одно небольшое препятствие. Функция `fromString` может отказать при извлечении некоторого числа, в этом случае она возвращает значение типа `option`, который определен следующим образом:

```
datatype 'a option = NONE | SOME of 'a;
```

Наконец, при выполнении последовательности операций ввода-вывода удобно помещать их в *выражение последовательности* (похожее на конструкцию `begin` из Scheme). В нем элементы помещены в круглые скобки и отделяются друг друга точками с запятой. Значением выражения последовательности считается значение последнего выражения в списке:

```
> fun printstuff () =
     ( output(stdOut,"Hello\n");
    output(stdOut,"World!\n")
    );
val printstuff = fn : unit -> unit
> printstuff ();
Hello
World!
val it = () : unit
```

Типы данных

В отличие от Scheme, язык ML предоставляет обширный набор типов данных, от перечисляемых типов до записей и списков. Списки и кортежи уже рассматривались. В то время как язык Scheme для моделирования других структур данных использует преимущественно списки, язык ML имеет много возможностей для создания типов данных, определяемых пользователем, в стиле языков C++ или Ada.

С использованием ключевого слова `type` здесь можно создавать синонимы существующих типов:

```
> type Money = real;
type Money = real
> 74000.00: Money;
val it = 74000.0 : Money
> type Location = real * real;
type Location = real * real
> (5.1,7.2): Location;
val it = (5.1,7.2) : Location
```

При помощи ключевого слова `datatype` можно вводить новые пользовательские данные. К примеру, перечисляемый тип можно определить следующим способом:

```
> datatype Go = Forward | Right | Back | Left;
```

Вертикальная черта используется для отделения альтернативных значений в объявлении и имен, таких как `Forward` и `Right`, называемых *конструкторами значений* или *конструкторами данных*. Конструкторы данных могут использоваться как паттерны в объявлениях или выражениях выбора, как в следующем объявлении функции:

```
> fun travel Forward = 0.0 |
    travel Right = 90.0 |
    travel Back = 180.0 |
    travel Left = 270.0 ;
val travel = fn : Go -> real
```

Рекурсивные типы, например бинарное дерево поиска, объявляются с помощью `datatype`:

```
> datatype 'a SearchTree = Nil | Node of 'a * 'a SearchTree * 'a SearchTree;
```

Данный тип параметризован типом данных, сохраняемых в дереве. Создадим дерево, соответствующее рис. 20.1:

```
> val tree = Node("man",
    Node( "girl", Nil, Node("liza",Nil,Nil) ),
    Node("boy",Node("tom",Nil,Nil), Nil));
val tree = Node ("man",Node ("girl",Nil,Node #),Node ("boy",Node #,Nil))
           : string SearchTree
```

Данные, а также функции `l-child` и `r-child` над деревом `SearchTree` можно определять путем сопоставления с образцом. К примеру:

```
> fun l-child (Node(data,left,right)) = left
    | l-child Nil = raise Empty;
val l-child = fn : 'a SearchTree -> 'a SearchTree
```

Функция обхода для бинарного дерева поиска, отображающая содержание строковых вершин, может быть представлена в следующей форме:

```
> fun display_tree Nil = () |
    display_tree (Node (data,left,right)) =
     (display_tree left;
        output(stdOut,data);
        output(stdOut,"\n");
        display_tree right);
val display_tree = fn : vector SearchTree -> unit
> display_tree tree;
girl
liza
man
tom
boy
val it = () : unit
```

Функции высшего порядка и карризация

В языке ML функции высшего порядка и выражения, чьими значениями являются функции, так же полезны, как и в языке Scheme. Для отображения тела функции в языке ML используется ключевое слово `fn` и специальная стрелка =>. К примеру:

```
> fn x => x * x;
val it = fn : int -> int
> (fn x => x * x) 5;
val it = 25 : int
```

Выражение `fn` применяется для создания анонимных функций и возвращаемых функциями величин. На самом деле определение функции `fun` (здесь и в Scheme) является просто синтаксическим сахаром для использования выражения `fn`.

Выражение

```
fun square x = x * x;
```

эквивалентно строке:

```
val square = fn x => x * x;
```

ПРИМЕЧАНИЕ

Термин «синтаксический сахар» используется для ссылки на механизм, который позволяет программисту заменять полную нотацию упрощенной, сокращенной нотацией. Например, оператор присваивания a += 5 является сокращением для эквивалентной, но более сложной формы a = a + 5 в языках C и Java.

Разница между ML и Scheme состоит в том, что если в ML мы используем выражение `fn` для объявления рекурсивной функции, то должны добавить ключевое слово `rec` (подобное слову `letrec` в языке Scheme):

```
val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

Для создания возвращаемых функцией величин, а также указания аргументов функций можно использовать функциональные выражения. К примеру, рассмотрим функцию `make_double`, которая повторяет аргумент для функции с двумя параметрами:

```
> fun make_double f = fn x => f (x,x);
val make_double = fn : ('a * 'a -> 'b) -> 'a -> 'b
> val square = make_double (op * );
val square = fn : int -> int
> val double = make_double (op +);
val double = fn : int -> int
> square 5;
val it = 25 : int
> double 10;
val it = 20 : int
```

Язык ML имеет ряд встроенных функций высокого порядка, включая функцию композиции, обозначаемую строчной буквой «o». К примеру:

```
> val doubleAndsquare = double o square;
val doubleAndsquare = fn : int -> int
> doubleAndsquare 5;
val it = 50 : int
```

Полезность функций высокого порядка и выражений, чьими значениями являются функции, существенно возрастает при использовании *карризации*. Говорят, что функция с множеством параметров, которая может быть представлена

как функция высокого порядка с одним параметром, возвращающая функцию с оставшимися параметрами, является *карризованной*. Это название дано в честь математика Хаскела Карри (Haskell B. Curry). Например, следующее определение функции `mapping` карризовано, поскольку параметры могут применяться отдельно (или вместе):

```
> fun mapping f [] = [] | mapping f (h::t) = (f h):: mapping f t;
val mapping = fn : ('a -> 'b) -> 'a list -> 'b list
> val squareList = mapping square;
val squareList = fn : int list -> int list
> squareList [5,6,7];
val it = [25,36,49] : int list
> mapping (fn x => x + x) [5,6,7];
val it = [10,12,14] : int list
```

В языке ML можно использовать кортеж для получения не карризованной версии функции или два отдельных параметра для получения карризованной версии:

```
(*не карризованная версия функции gcd: *)
> fun gcd (u,v) = if v = 0 then u else gcd (v , u mod v);
val gcd = fn : int * int -> int
(*карризованная версия функции gcd: *)
> fun gcd u v = if v = 0 then u else gcd v (u mod v);
val gcd = fn : int -> int -> int
```

В языке ML не карризованная версия функции, использующая кортеж, в действительности является функцией с единственным параметром-кортежем.

Язык называют *полностью карризованным*, если все определения функций автоматически считаются карризованными, а также все встроенные функции с несколькими параметрами (такие как арифметические операции) являются карризованными. Язык ML не является полностью карризованным, поскольку все его встроенные бинарные операции определены как принимающие кортежи:

```
> op +;
val it = fn : int * int -> int
> op ::;
val it = fn : 'a * 'a list -> 'a list
> op @;
val it = fn : 'a list * 'a list -> 'a list
```

Можно определить карризованные версии этих функций, но все они превратятся из операций в префиксные функции:

```
> fun plus x y = x + y;
val plus = fn : int -> int -> int
> plus 1 2;
val it = 3 : int
> val plusTwo = plus 2;
val plusTwo = fn : int -> int
> plusTwo 5;
val it = 7 : int
> fun append L1 L2 = L1 @ L2;
val append = fn : 'a list -> 'a list -> 'a list
> val appendToOne = append [1];
val appendToOne = fn : int list -> int list
> appendToOne [7,9];
val it = [1,7,9] : int list
```

Отложенные вычисления

Важной проблемой функциональных языков считают различие между обычными применениями функций и специальными формами. В языке с правилом, определяющим порядок вычислений, все параметры для пользовательских функций вычисляются во время вызова, даже если в этом нет необходимости.

Типичными примерами, которые не используют правило порядка вычислений, являются специальные булевы формы `and`, `or` и `if`. В случае специальной формы `and` в выражении на языке Scheme `(and a b)` не требуется вычислять параметр `b`, если вычисление `a` привело к `false`. Это называется короткой схемой вычисления булевых выражений и демонстрирует полезность отложенных вычислений. В специальной форме `if` отложенность не просто полезна, а совершенно необходима: чтобы в языке Scheme выражение `(if a b c)` имело правильную семантику, вычисление `b` и `c` должно быть отложено до тех пор, пока не станет известен результат вычислений `a`, после чего вычисляется или `b`, или `c`, но не оба элемента. По этой причине специальная форма `if` не может быть определена в качестве стандартной функции языка Scheme или ML. Это также означает, что в Scheme и ML следует различать два вида форм, одна из которых использует стандартные вычисления (применения функций), а другая — нет (специальные формы).

Нестрогие функции

Целесообразность ограничений в ходе вычислений обусловлена возможным упрощением семантики и реализации. Обсудим выражение, в котором значение подвыражения может быть не определено, как, например, в выражении, записанном на языке Scheme `(and (= 1 0) (=1 (/ 1 0)))`, или в его эквиваленте на языке C `(1 == 0) && (1 == 1 / 0)`. В этом случае отложенные вычисления могут привести к правильному результату, даже если подвыражения или параметры не определены. Подобные функции называют *нестрогими*. Строгие языки (языки со строгими функциями) проще для реализации. По сути, строгие языки соответствуют сильной форме принципа GIGO (garbage in, garbage out), постоянно отказываясь вырабатывать результаты при неполных и неправильных входных данных. Кроме того, строгость оказывает важное влияние на упрощение формальной семантики.

Тем не менее, как вы видели, нестрогость может быть желаемым свойством. Нестрогие функции могут вырабатывать результаты даже в присутствии неполных и неправильных входных данных: главное, чтобы информации было достаточно для формирования разумного результата. Интересно отметить, что язык Algol 60 применял отложенное вычисление при передаче параметра по имени. Здесь параметр вычислялся только тогда, когда он фактически использовался в коде вызываемой процедуры. Таким образом, функция с передачей по имени:

```
function calc(a: boolean; x: integer): integer;
begin
    if a then calc := 1
    else calc := x;
end;
```

возвращает значение 1 при вызове calc(true, 1 div 0), так как параметр x в теле функции calc достигнут не будет и, следовательно, значение x — не определяемое выражение 1 div 0 — вычисляться не будет.

В языке с поддержкой «функций как значений» для задержки вычисления параметр можно поместить в функцию-оболочку (функцию без параметров). К примеру, в языке C эффекта передачи параметра по имени из предыдущего примера можно достичь, записав:

```
typedef int (*IntFunc) ();
int divByZero ()
{ return 1 / 0;
}
int calc(int a, IntFunc x)
{ if (a) return 1;
    else return x();
}
```

и вызвав calc(1,divByZero). В языках Scheme и ML разместить параметры в функциональных оболочках даже проще, поскольку лямбда-конструкторы и конструкторы функциональных значений fn могут использоваться непосредственно, как в следующем примере на языке Scheme:

```
(define (calc a x) (if a 1 (x)))
```

который можно вызвать следующим образом:

```
(calc #T (lambda () (/ 1 0)))
```

Заметим, что в теле функции calc учтено, что параметр x является функцией: он окружен круглыми скобками, что инициирует вызов x. В противном случае будет возвращаться сама функция, а не ее значение.

Специальные формы для отложенных вычислений

Язык Scheme содержит две специальные формы, реализующие то, что мы описали. Специальная форма delay откладывает вычисления ее аргументов и возвращает объект, который можно рассматривать как лямбда-оболочку или как обещание вычислить аргументы формы. Соответствующая специальная форма force вызывает вычисление ее параметра, который должен быть отложенным объектом (будем называть это *форсированием*). Таким образом, функция calc записывается в виде:

```
(define (calc a x) (if a 1 (force x)))
```

и вызывается:

```
(calc #T (delay (/ 1 0)))
```

Эффективность отложенных вычислений понижается при повторных вычислениях отложенного выражения. Например, в отложенной версии функции square

```
(define (delayedSquare x) (* (force x) (force x)))
```

параметр x дважды вычисляется в теле delayedSquare, что снижает эффективность, если x является сложным выражением. На деле язык Scheme улучшает ситуацию, встраивая в функцию force передачу по имени.

Язык Scheme использует процесс *мемоизации*: при первом применении `force` к отложенному объекту вычисленное значение сохраняется, а при последующих вызовах `force` просто возвращается сохраненное значение. Такой вид передачи и вычисления параметра называют *передачей по необходимости*.

Полезность отложенных вычислений при передаче по имени заключается в следующем:

- параметры остаются не вычисленными до тех пор, пока в этом нет необходимости;
- поддерживается определение специальных форм (`if` и `cond`) в виде обычных функций.

Тем не менее передача по имени не способна обеспечить более сложную ситуацию, когда требуется вычисление лишь части каждого параметра. Рассмотрим функцию `select_part`, которая возвращает первые `n` элементов списка:

```
(define (select_part n L)
    (if (= n 0) '()
        (cons (car L) (select_part (- n 1) (cdr L)))))
```

Если мы запишем версию, в которой вычисление `L` отложено

```
(define (select_part n L)
    (if (= n 0 ) '()
        (cons ( car (force L)) (select_part (- n 1)
            (cdr (force L))))))
```

тогда вызов `(select_part 1 (delay L))` приведет к вычислению целого списка `L`, хотя нужен лишь первый элемент. Это очень неэффективно, особенно если список достаточно длинный и создается следующей процедурой генерации:

```
(define (list_int m n)
    (if (> m n) '() (cons m (list_int (+ 1 m) n))))
```

При вызове `(select_part 1 (delay (list_int 2 100)))` будет конструироваться целый список `(2..100)`, и лишь после этого будет взят первый элемент `( 2 )`. Нужно применить `delay` ко второму параметру функции `cons` в теле `list_int`:

```
(define (list_int m n)
(if (> m n) '() (cons m (delay (list_int (+ 1 m) n)))))
```

вследствие чего взятие `(cdr (cons m (delay ...)))` вернет отложенный объект в рекурсивный вызов функции `select_part`. Таким образом, в этом вычислении имеется следующая последовательность событий, где каждый отложенный объект представляется вычислением его обещания выполнить то, что заключено в кавычки:

1. Вызов `(select_part 1 (delay (list_int 2 100)))` приводит к передаче отложенного объекта `(list_int 2 100)` в список `L` для:

   ```
   (cons (car (force L)) (select_part (- 1 1) (cdr (force L))))
   ```

2. Первый вызов `(force L)` в функции `cons` приводит к вычислению над `L`:

   ```
   (cons 2 ((delay (list_int (+ 1 2) 100))))
   ```

   которое вызывает создание списка с головой 2 и хвостом в виде отложенного объекта

   ```
   (list_int (+ 1 2) 100)
   ```

3. Вызов `car` над `(force L)` вернет `2` из `cons`, оставив следующее выражение для вычисления

```
(cons 2 (select_part (- 1 1) (cdr (force L))))
```

4. Вызов `(select_part (- 1 1) (cdr (force L)))` приведет к вычислению для `(- 1 1)` величины `0`, а вызов `cdr` над `(force L)` вычислит отложенный объект `(list_int (+ 1 2) 100)` как созданный на шаге 2. Выражение `(select_part 0 (list_int (+ 1 2) 100))` вычисляется и возвращает пустой список `'()` без инициализации вычисления `(list_int (+ 1 2) 100)`.
5. В конечном счете вычисляется результат `(cons 2 '())`, возвращая список `(2)`. Следовательно, остаток целого списка никогда не вычисляется.

Ленивые вычисления

Описанный выше сценарий, вместе с мемоизацией, называется *ленивыми вычислениями*. Этот механизм вычислений можно реализовать в функциональном языке без явных вызовов `delay` и `force` при соблюдении средой управления вычислениями следующих правил:

- Все аргументы пользовательских функций являются отложенными.
- Все связывания локальных имен в выражениях `let` и `letrec` являются отложенными.
- Все аргументы конструкторских функций (такие как `cons`) являются отложенными.
- Все аргументы остальных предопределенных функций, таких как арифметические функции «*» и т. д., являются форсируемыми.
- Все аргументы-функции являются форсируемыми.
- Все условия в формах ветвления, таких как `if` и `cond`, являются форсируемыми.

Согласно этим правилам операции над длинными списками производят вычисления лишь по мере необходимости. Ленивые вычисления делают возможной обработку потенциально бесконечных списков, поскольку лишь часть списка может быть затребована в некий момент времени. Списки, которые подчиняются правилам ленивых вычислений, часто называются потоками. Это название наводит на мысль о бесконечной природе списков. Поток считается частично вычислимым списком, чьи элементы определены вплоть до любого требуемого количества. Потоки являются чрезвычайно важным механизмом функционального программирования. Для полного устранения побочных эффектов все вводы-выводы функциональных программ должны быть организованы в виде потоков. Языки Scheme и ML содержат специальные потоковые конструкторы.

Основным примером функционального языка с ленивыми вычислениями является Haskell. В среде ленивых вычислений все функции делятся на генераторы и фильтры. Функции, которые генерируют потоки, называются генераторами, а функции, модифицирующие потоки, именуют фильтрами. К примеру, функция

`list_int` из последнего примера является генератором, а функция `select_part` — фильтром

Ленивые вычисления используются не во всех функциональных языках, поскольку при этом усложняется семантика языка и среда управления вычислениями, которая должна реализовать перечисленные выше правила. Кроме того, из-за чередования задержек (`delay`) и форсирований (`force`) трудно разрабатывать программы с желаемыми сторонними эффектами, например программы с переменными, которые изменяются по мере вычисления доходов. В некотором смысле ситуация похожа на проблему синхронизации для разделяемой памяти при параллельных вычислениях. Отложенные вычисления можно описывать как форму параллелизма, задержку, как приостановку процессов, а форсирование как возобновление процесса. Главное состоит в том, что побочные эффекты (в частности, присваивание) плохо сочетаются с ленивыми вычислениями. Это объясняет, почему Haskell является чисто функциональным языком, без переменных и присваивания, а языки Scheme и ML считаются строгими языками с некоторыми потоковыми возможностями и средствами для отложенных вычислений.

Контрольные вопросы и упражнения

1. Дайте определение выражения и специальной формы. Чем они схожи? В чем отличаются?
2. Напишите функцию на языке Scheme, возвращающую список, составленный в обратном порядке по сравнению с простым списком, являющимся ее параметром.
3. Напишите предикатную функцию на языке Scheme для проверки структурной эквивалентности двух заданных списков. Два списка называются структурно эквивалентными, если они имеют одинаковую структуру, несмотря на то что их атомы могут отличаться друг от друга.
4. Напишите функцию на языке Scheme, возвращающую объединение двух простых списков, являющихся ее параметрами и представляющими собой множества.
5. Напишите на языке Scheme функцию с двумя параметрами, атомом и списком, возвращающую список, из которого удален заданный атом, независимо от глубины его нахождения в списке. Возвращаемый список не должен иметь на месте удаленного атома никакого элемента.
6. Напишите на языке Scheme функцию, которая получает список в качестве параметра и возвращает его, удалив из него второй от начала элемент. Если заданный список не содержит хотя бы двух элементов, функция должна возвращать пустой список ().
7. Назовите главные особенности, которые отличают язык ML от языка Scheme.
8. Напишите программу на языке ML для поиска совершенных чисел; $n > 2$ — совершенное число, если множители (не включая n) в сумме дают n. Например, $1 + 2 + 4 + 7 + 14 = 28$.
9. Напишите функцию, которая берет список деревьев и возвращает список минимальных значений в каждом дереве.

10. Создайте следующие функциональные программы с помощью языков Scheme и ML:
    1) Напишите функцию с хвостовой рекурсией для вычисления длины произвольного списка.
    2) Напишите функцию, которая вычисляет максимум и минимум списка из целых чисел.
    3) Напишите функцию, которая собирает целые числа, вводимые пользователем, подсчитывает их количество и возвращает их в виде списка, где они располагаются в порядке ввода.
    4) Используя функции из упражнений (2) и (3), напишите программу для ввода списка целых чисел, заканчивающегося нулем, печати списка в порядке ввода и печати максимального и минимального из чисел.
    5) Напишите сортировку Quicksort для списка целых чисел.
    6) Напишите сортировку Mergesort для списка целых чисел.
    7) Напишите функцию высшего порядка `twice`, которая принимает как параметр функцию с одним аргументом и возвращает функцию, которая представляет применение этой функции к ее аргументу дважды. Для обычного определения функции `square` что означает функция `(twice (twice square))`?
    8) Напишите функцию высшего порядка `incN`, которая принимает целое *n* как параметр и возвращает функцию *n*-го инкрементирования, которая увеличивает ее параметр на *n*. Таким образом, в синтаксисе языка Scheme: `((incN 3) 2) = 5 and ((incN -2) 3) = 1`.

Глава 21
Логическое программирование

Цель этой главы — ввести основные понятия логического программирования и познакомить с базовыми средствами языка Prolog. Вначале рассмотрим математические основы логического программирования: исчисление предикатов. Затем обсудим трудности, возникающие при использовании исчисления предикатов в автоматизированных дедуктивных системах, и покажем пути их решения, акцентировав внимание на формулах Хорна и механизме «резолюции–унификации». Потом сделаем общий обзор языка логического программирования Prolog, поясняя его нотацию и структуры данных, процесс выполнения программы, арифметику, унификацию и стратегию поиска, а также циклы и структуры управления. В завершение опишем некоторые проблемы, связанные с применением языка Prolog в логическом программировании: проверку вхождения при унификации, замкнутость мира, своеобразие логического отрицания, ограниченность формул Хорна при описании логики утверждений, принудительное внесение информации управления.

Исчисление предикатов

Прежде чем обсуждать логическое программирование, следует познакомиться с математической логикой. Разновидность логики, используемой в логическом программировании, называется *исчислением предикатов первого порядка*. Это исчисление — удобный способ формального выражения логических утверждений, то есть утверждений, которые или верны, или ложны.

Например, следующие фразы являются логическими утверждениями:

```
0 — это натуральное число.
2 — это натуральное число.
Для всех x, если x — натуральное число, то существует число, следующее за x.
141 — это натуральное число.
```

В терминах исчисления предикатов они принимают вид:

```
natural (0).
natural (2).
For  all  x. natural (x ) natual (successor (x )).
natural (141).
```

Среди приведенных логических утверждений первое и третье утверждения могут рассматриваться как аксиомы для натуральных чисел: утверждения, которые

приняты как истинные и с помощью которых могут быть доказали все истинные утверждения о натуральных числах. Действительно, второе утверждение может быть доказано из этих аксиом, так как

```
2 = successor (successor (0)) and natural (0) natural (successor(0)) natural (suc-
cessor (successor (0)))
```

Напротив, может показаться, что четвертое утверждение нельзя доказать на основе аксиом и поэтому оно может быть принято как ложное.

Компоненты исчисления предикатов первого порядка

Опишем основные компоненты исчисления предикатов первого порядка, которые определяют любые логические утверждения.

Константы. Константами являются числа или имена. Иногда их называют атомами, так как они не могут быть разбиты на части. Например, число 0 — это константа.

Предикаты. Они именуют функции, которые могут быть истинными или ложными, и подобны булевым функциям в императивных программах. Предикаты могут иметь несколько аргументов. Предикат `natural(0)` имеет один аргумент.

Функции. Исчисление предикатов первого порядка различает функции, которые верны или ложны — это предикаты — и все другие функции, подобные `successor (x)`, которые вычисляют не булевы величины.

Переменные, которые представляют еще не определенные величины. Например, `x` — переменная.

Соединители. Они включают операции `and`, `or` и `not` (точно такие же, как и операции над булевыми данными в языках программирования). Дополнительными соединителями в исчислении предикатов считаются импликация, показываемая как однонаправленная стрелка →, и эквивалентность, показываемая как двунаправленная стрелка ↔. В действительности это не новые операции: `a` → `b` означает, что элемент `b` верен всякий раз, когда верен элемент `a`. Эквивалентным является утверждение `b or not a`. Также и `a` ↔ `b` означает то же самое, что `(a` → `b) and (b` → `a)`.

Кванторы. Кванторы обозначают операции, которые задают специфику применения переменных. Фраза `For all x` — квантор для `x`; его называют *квантором всеобщности*. Утверждение

```
For  all  x, natural (x ) natual (successor (x ))
```

означает, что для всех `x`, если `x` — натуральное число, то следующее за `x` число также является натуральным числом. Квантор всеобщности заявляет, что отношения среди предикатов верны для всех значений переменной `x`.

Кроме того, имеется *квантор существования* `there exists`, как записано в следующем утверждении:

```
there exists x,   natural (x)
```

Это утверждение означает, что существует такое значение переменной `x`, при котором `x` является натуральным числом. Квантор существования позволяет заявить, что предикат верен по крайней мере для одного значения переменной `x`.

Говорят, что кванторы связывают переменные. Переменные, для которых кванторы не определены, называют свободными.

Символы пунктуации. Они включают левые и правые круглые скобки, запятую и точку. Строго говоря, точка вовсе не обязательна, но ее использует большинство систем логического программирования. Круглые скобки применяются к включаемым аргументам и группировкам операций. Круглые скобки могут опускаться в соответствии с соглашениями о приоритетах соединителей, которые обычно принимаются такими же, как и в большинстве языков программирования. Перечислим соединители в порядке уменьшения их приоритета: `not`, `and`, `or`, `→`, `↔`.

В исчислении предикатов аргументами предикатов и функций могут быть только *термы*, то есть комбинации переменных, констант и функций. Термы не могут содержать предикаты, кванторы или соединители. Термами являются константы `0`, `141`, `2`, переменная `x`, а также фраза `successor(x)`, которая включает функцию `successor` с аргументом `x`. Дополнительными примерами термов могут быть `successor(0)` и `successor(successor(successor(x)))`.

Запись утверждений в исчислении предикатов

Рассмотрим логические утверждения:

```
1.    Лошадь – млекопитающее.
2.    Человек – млекопитающее.
3.    У млекопитающих есть четыре ноги и отсутствуют руки, или две ноги и две руки.
4.    У лошади нет рук.
5.    У человека есть руки.
6.    У человека нет ног.
```

Переведем эти утверждения в термины исчисления предикатов:

```
1.    млекопитающее (лошадь).
2.    млекопитающее (человек).
3.    for all x, млекопитающее (x)
       ноги (x, 4) and руки (x, 0) or ноги (x, 2) and руки (x, 2).
4.    руки (лошадь, 0).
5.    not руки (человек, 0).
6.    ноги (человек, 0).
```

В данном примере константы — это целые числа `0`, `2` и `4`, а также имена `лошадь` и `человек`. Предикаты — `млекопитающее`, `руки` и `ноги`. Единственная переменная — `x`, и нет никаких функций.

Первые пять утверждений следует считать аксиомами, определяющими истинные отношения. Легко видеть, что на основе этих аксиом доказывается `руки (человек, 2)`. Также можно доказать, что верно `ноги (человек, 2)`. Шестое же утверждение при учете аксиом ложно.

Для минимизации круглых скобок в записях утверждений использовались приоритеты, поскольку, например, когда мы пишем

```
ноги (x, 4) and руки (x, 0) or ноги (x, 2) and руки (x, 2)
```

то подразумевается следующее:

```
(ноги (x, 4) and руки (x, 0)) or (ноги (x, 2) and руки (x, 2))
```

Правила вывода исчисления предикатов первого порядка

Введенные компоненты исчисления предикатов следует дополнить правилами вывода. Правила вывода производят или доказывают новые утверждения на основе имеющегося набора утверждений.

Типичное правило вывода имеет следующий вид:

```
из утверждений a → b и b → c можно получить утверждение a → c.
```

Формально же данное правило вывода записывается следующим образом:

$$\frac{(a \to b) \text{ and } (b \to c)}{a \to c}.$$

Правила вывода позволяют строить набор всех утверждений, которые могут быть произведены или доказаны на основе исходного набора утверждений: такие утверждения имеют истинные значения, если верны оригинальные утверждения. Например, первые пять утверждений о млекопитающих (из предыдущего подраздела) позволяют получать следующие утверждения:

```
ноги (лошадь, 4).
ноги (человек, 2).
руки (человек, 2).
```

Заявленные на языке логики, эти три утверждения становятся теоремами, полученными из первых пяти утверждений (или аксиом) о млекопитающих. Заметьте, что доказательство утверждений (на базе исходных утверждений) может рассматриваться как вычисление числа рук и ног лошади или человека. Таким образом, набор исходных утверждений задает потенциальные вычисления всех логических последствий данных утверждений.

В этом заключается сущность логического программирования: набор утверждений принимается в качестве аксиом, а желаемый факт производится путем автоматического применения к аксиомам правил вывода. Таким образом, следует определить:

Язык логического программирования — это система для записи логических утверждений вместе с определенными алгоритмами для реализации правил вывода.

Набор логических утверждений в качестве аксиом может рассматриваться как логическая программа, а утверждения о производстве результатов считаются вводами, которые инициируют вычисления. Такие вводы предоставляются программистами и называются запросами или целями. Например, для набора аксиом о млекопитающих, при желании узнать, сколько ног у человека, мы обеспечили бы следующий запрос:

```
Существует ли такое значение y, которое равно числу ног у человека?
```

или, в терминах исчисления предикатов:

```
there exists  y,  ноги (человек, y)?
```

а система ответила бы примерно так:

```
да: y 2
```

Поэтому системы логического программирования иногда называют дедуктивными базами данных, то есть базами данных, состоящими из набора утверждений и дедуктивной системы, которая может отвечать на запросы. Заметьте, что они отличаются от обычных баз данных, так как содержат не только факты, такие как `млекопитающее (человек)` или `natural (0)`, но также и более сложные утверждения, такие как `natural(x) natural (successor (x))`. Кроме того, система может отвечать не только на запросы по фактам, но и на запросы, требующие логических выводов.

Специфика логического программирования

В системе чисто логического программирования ничего не сказано о том, как конкретное утверждение должно быть произведено из некоторого набора утверждений. Определенный путь или последовательность шагов, которые автоматическая дедуктивная система выбирает для производства утверждения, является проблемой управления для системы логического программирования. Обычные утверждения представляют логику вычисления, в то время как дедуктивная система обеспечивает управление доказательством нового утверждения. Это свойство систем логического программирования принудило Р. Ковальски декларировать парадигму логического программирования как псевдоуравнение:

```
алгоритм = логика + управление,
```

что контрастирует с выражением Н. Вирта для императивного программирования:

```
алгоритмы = структуры данных + программы
```

Принцип Ковальски указывает на существенную особенность логического программирования: так как логические программы не отражают вопросы управления, операции (в теории, по крайней мере) могут выполняться в любом порядке (или одновременно). Таким образом, языки логического программирования — наиболее подходящие кандидаты для распараллеливания вычислений.

К сожалению, автоматизированные дедуктивные системы испытывают трудности при непосредственном использовании исчисления предикатов первого порядка. Во-первых, одни и те же утверждения можно выразить огромным количеством способов. Во-вторых, существует слишком много правил вывода. В этих условиях большинство систем логического программирования вынуждено ограничивать себя специфическим подмножеством исчисления предикатов, названного *формулами (дизъюнктами) Хорна*, к изучению которых мы переходим.

Формулы Хорна

Формула Хорна (названная в честь ее изобретателя Альфреда Хорна) является утверждением вида:

a_1 and a_2 and a_3... and $a_n \rightarrow b$

где a_i позволено быть простым утверждением, не содержащим соединителей. Таким образом, в формулах Хорна нет соединителей `or` и кванторов. Предыдущая

формула Хорна говорит, что утверждения с a_1 по a_n совместно имплицируют b или что b — истина, если все a_i имеют истинные значения. Элемент b называют *головой* формулы, а a_1 ,..., a_n — это *тело* формулы. В формуле Хорна количество a_i может быть равно нулю, тогда формула Хорна принимает вид:

```
→ b
```

Данная формула означает, что b всегда является истиной. Другими словами, b — это аксиома и обычно записывается без соединителя →. Такие формулы называют *фактами*.

Формулы Хорна могут выражать большинство, но не все логические утверждения. Существует алгоритм трансляции утверждений исчисления предикатов в формулы Хорна. Основная идея состоит в том, чтобы удалять соединители or за счет написания отдельных утверждений, а также трактовать отсутствие кванторов как предположение, что переменные, появляющиеся в голове формулы, связаны квантором всеобщности, в то время как переменные, появляющиеся в теле формулы (но не в голове), связаны квантором существования.

Например, следующие утверждения

```
natural (0).
For all x, natural (x) natural (successor (x)).
```

могут быть транслированы в формулы Хорна удалением квантора:

```
natural (0).
natural (x) → natural (successor (x)).
```

Рассмотрим логическое описание алгоритма Евклида для вычисления наибольшего общего делителя (НОД) двух положительных целых чисел u и v:

```
НОД u и 0 – это u.
НОД u и v, если v не равно 0, равен НОД v и остатка от деления v на u.
```

Транслируем данное описание в термины исчисления предикатов первого порядка:

```
for all u, НОД (u, 0, u).
for all u, for all v, for all w,
    not zero (v) and НОД (v, u mod v, w) НОД (u, v, w).
```

(Помним, что НОД (u, v, w) — это предикат, выражающий, что w — это НОД для u и v.)

При трансляции этих утверждений в формулы Хорна мы должны снова удалить кванторы:

```
НОД (u, 0, u).
not zero (v) and НОД (v, u mod v, w) → НОД (u, v, w).
```

Предыдущие примеры содержали лишь переменные, связанные квантором всеобщности. Продемонстрируем обработку переменной из тела, связанной квантором существования. Рассмотрим следующее утверждение:

```
x – прародитель y, если x – родитель кого-то, кто является родителем y
```

Транслируя утверждение в исчисление предикатов, мы получим

```
for all x, for all y, (there exists  z, parent (x, z) and grandparent (z, y))
    → grandparent (x, y).
```

Переходя к формуле Хорна, запишем:

```
parent (x, z) and parent (z, y) → grandparent (x, y)
```

Для демонстрации обработки соединителей рассмотрим следующее утверждение

```
for all x, если x – млекопитающее, тогда у x есть две или четыре ноги
```

Транслируя в исчисление предикатов, получим:

```
for all x, млекопитающее (x) ноги (x, 2) or ноги (x, 4)
```

Это может быть аппроксимировано следующими формулами Хорна:

```
млекопитающее (x) and not ноги (x, 2) → ноги (x, 4).
млекопитающее (x) and not ноги (x, 4) → ноги (x, 2).
```

В общем случае, чем больше соединителей появляется справа от соединителя →, тем тяжелее их транслировать во множество формул Хорна.

Формулы Хорна особенно интересны для автоматических дедуктивных систем, таких как системы логического программирования, поскольку они обеспечивают процедурную интерпретацию. Если записать формулу Хорна в обратном порядке

$b \leftarrow a_1$ and a_2 and a_3... and a_n

ее можно рассматривать как определение процедуры b: тело b образуется телом формулы, а именно, операциями, которые указаны как a_i. Это очень похоже на способ, которым правила контекстно-свободной грамматики интерпретируют определения процедуры при рекурсивном нисходящем разборе. Более того, логические программы могут использоваться для прямого создания синтаксических анализаторов. Фактически синтаксический анализ естественного языка был одним из мотивов для первоначальной разработки языка Prolog. Главное различие между синтаксическим анализом и логическим программированием состоит в том, что в чисто логическом программировании порядок вызова a_i не определен.

Тем не менее большинство систем логического программирования детерминировано в том, они выполняют вызовы в определенном порядке (обычно слева направо), который точно соответствует порядку, задаваемому правилами контекстно-свободной грамматики. Специфический вид правил грамматики, используемых в программах языка Prolog, называют *грамматиками определенных формул*.

Формулы Хорна следует воспринимать как спецификации процедур, а не как их реализации. Например, следующая формула Хорна является спецификацией процедуры сортировки:

```
sort (x, y) ← перестановка (x, y) and sorted (y)
```

Здесь говорится, что (предположим, что x и y — некоторые списки элементов) процедура сортировки преобразует список x в отсортированный список y, для которого y — это перестановка x. Для завершения общей спецификации нужно, конечно, предоставить спецификации ее элементов, определяющие смысл сортировки списка и смысл перестановки списка.

Важно то, что мы думаем о формуле Хорна не как о необходимом алгоритме, который должен быть найден для y при заданном x, а как о свойствах, которые должен иметь список y.

Исходя из процедурной интерпретации, большинство систем логического программирования не только записывает формулы Хорна в обратном порядке,

но и устраняет соединители `and` между a_i, заменяя их запятыми. Таким образом, введенные утверждения для НОД трансформируются следующим образом:

```
НОД (u, 0, u).
НОД (u, v, w) ← not zero (v), НОД (v, u mod v, w)
```

Можно заметить, что приведенная запись очень похожа на выражение для НОД в терминах стандартного языка программирования:

```
НОД (u, v) = if v = 0 then u else НОД (v, u mod v)
```

Теперь мы будем записывать формулы Хорна только в этой форме.

Рассмотрим проблему области видимости переменной в процедурной интерпретации формул Хорна. Допустим, что в каждом вызове процедуры все переменные являются локальными. (Это соответствует определениям процедуры в языке с блочной структурой.) Действительно, переменные, используемые в голове, могут рассматриваться как параметры, в то время как переменные, используемые только в теле, могут рассматриваться как локальные, временные переменные. Алгоритмы, реализующие исполнение формул Хорна, при обработке переменных исходят из подобных соображений. Более общую картину представим в следующих разделах.

В системах логического программирования формулы Хорна, содержащие как голову, так и тело, называют *правилами*. Правилом является, например, приведенная ранее формула:

```
ноги (x, 2) ← млекопитающее (x), not ноги (x, 4)
```

Между правилами и фактами существует важное различие.

Такие факты, как

```
родитель (сергей, иванна)
```

представляют собой логические утверждения, которые всегда и безусловно являются истинными.

С другой стороны, правила — это утверждения, которые становятся истинными, если удовлетворяются некоторые условия.

Условия образуют утверждения из правой части правила — его тела. Голова правила содержит заключение, истинность которого является логическим следствием истинности условий. В приведенном выше примере заключение о наличии у `x` двух ног становится истиной, если `x` является млекопитающим и у него нет четырех ног.

Зададим вопрос: как выражаются в формулах Хорна утверждения запросов или целей?

По сути, запрос точно противопоставляется факту, являясь формулой Хорна без головы:

```
млекопитающее (человек) ←. — Это факт
млекопитающее (человек). — Это запрос или цель
```

Слово «цель» (goal) используется потому, что система логического программирования воспринимает это утверждение как цель, которую необходимо достичь.

Формула Хорна без головы может включать последовательность запросов, отделяемых запятыми:

```
млекопитающее (x), ноги (x, y)
```

Почему запросы соответствуют формулам Хорна без головы, становится ясно, когда вы понимаете суть правила вывода. Ведь правило вывода в систе-

ме логического программирования применяется для доказательства новых утверждений на основе набора заданных утверждений и согласовано с принципом резолюции.

Резолюция и унификация

Резолюция — это правило вывода для формул Хорна, которое делает вывод особенно эффективным. Резолюция говорит, что если имеются две формулы Хорна и найдено соответствие головы первой формулы Хорна одному из утверждений в теле второй формулы, тогда тело первой формулы может заменить соответствующее утверждение в теле второй формулы. Если у нас есть формулы Хорна:

$a \leftarrow a_1, \ldots, a_n.$
$b \leftarrow b_1, \ldots, b_m.$

и b_i соответствует a, тогда можно вывести формулу:

$b \leftarrow b_1, \ldots, b_{i-1}, a_1, \ldots, a_n, b_{i+1}, \ldots, b_m$

Самый простой пример такого сопоставления имеет место, когда тела формул Хорна содержат только по одному утверждению:

$b \leftarrow a$

и

$c \leftarrow b$

В этом случае резолюция позволяет вывести следующую формулу:

$c \leftarrow a$

Другой подход в резолюции состоит в объединении левых и правых частей обеих формул Хорна и дальнейшем вычеркивании тех утверждений, которые появляются в обеих частях новой формулы. Таким образом, для простейших формул:

$b \leftarrow a$

и

$c \leftarrow b$

получаем:

$b, c \leftarrow a, b$

и, вычеркивая b,

$\cancel{b}, c \leftarrow a, \cancel{b}$

выводим:

$c \leftarrow a$

Теперь обсудим, как система логического программирования может обработать цель или список целей в виде формул Хорна без головы. Система пытается применить резолюцию, сопоставляя одну из целей в теле формулы без головы с головой известной формулы. Затем она заменяет сопоставленную цель телом этой формулы, создавая новый список целей, которые она продолжает модифицировать таким же образом. Новые цели называют *подцелями*. Если у нас есть цель:

$\leftarrow a$

а также формула $a \leftarrow a_1, \ldots, a_n$, тогда резолюция заменяет оригинальную цель подцелями:

$\leftarrow a_1, \ldots, a_n$

Если системе удается в конечном счете устранить все цели, то есть произвести пустую формулу Хорна, тогда оригинальное утверждение считается доказанным.

Проиллюстрируем сказанное. В простейшем случае цель является уже известным фактом, например, таким:

```
млекопитающее (человек)
```

Положим, что некто спрашивает — является ли человек млекопитающим:

```
← млекопитающее (человек)
```

Используя резолюцию, система объединяет две формулы Хорна в одну:

```
млекопитающее (человек) ← млекопитающее (человек)
```

А затем, вычеркивая одинаковые утверждения в обеих частях, получает:

```
←
```

Таким образом, система нашла, что действительно человек — это млекопитающее, и ответила бы на запрос «Да».

Обсудим более сложный пример. Даны правила и факты:

```
ноги(x, 2) ← млекопитающее(x), руки(x, 2).
ноги(x, 4) ← млекопитающее (x), руки(x, 0).
млекопитающее (лошадь).
руки(лошадь, 0).
```

Если мы сформируем запрос:

```
← ноги(лошадь,4)
```

Тогда, применяя резолюцию, использующую второе правило, получим:

```
ноги(x, 4) ← млекопитающее (x), руки(x, 0), ноги(лошадь, 4)
```

Теперь для вычеркивания утверждений, применяющих предикат `ноги`, из обеих частей мы сопоставляем переменную `x` с `лошадью`, поэтому заменим `x` на `лошадь` во всех утверждениях:

```
ноги(лошадь, 4) ← млекопитающее(лошадь), руки(лошадь, 0), ноги(лошадь, 4)
```

Далее вычеркиваем для получения подцелей:

```
← млекопитающее (лошадь), руки(лошадь, 0)
```

Теперь применим резолюцию дважды, используя факты `млекопитающее(лошадь)` и `руки(лошадь, 0)`, а также вычеркивание:

```
млекопитающее(лошадь) ← млекопитающее(лошадь), руки(лошадь, 0).
руки(лошадь, 0).
руки(лошадь, 0) ← руки(лошадь, 0).
```

Так как мы пришли к пустому утверждению, наш оригинальный запрос истинен.

Последний пример демонстрирует дополнительную технику, применяемую в резолюции для производства целей.

Чтобы сопоставлять утверждения, которые содержат переменные, следует устанавливать переменные равными термам так, чтобы утверждения становились идентичными и могли подвергнуться вычеркиванию с обеих сторон. Этот процесс *сопоставления с образцом* (для обеспечения идентичности утверждений) называют

унификацией, а переменные, которые устанавливаются равными образцам, называют *конкретизированными*.

Таким образом, для обеспечения эффективной реализации резолюции необходимо разработать алгоритм для унификации. Такой алгоритм унификации мы обсудим в следующем разделе.

Для более детального представления разновидности унификации, используемой в языке логического программирования, обсудим проблему наибольшего общего делителя (введенный ранее алгоритм Евклида):

```
нод(u, 0, u).
нод(u, v, w) ← not zero(v), нод(v, u mod v, w).
```

Например, для заданной цели:

```
← нод(15, 10, x)
```

при использовании первой формулы резолюция отказывает (`10` не сопоставляется с `0`), однако использование второй формулы и унификации `нод(u, v, w)` с `нод(15, 10, x)` дает:

```
нод(15, 10, x) ← not zero(10), нод(10, 15 mod 10, x), нод(15, 10, x)
```

Допустим, что система знает, что `zero(10)` ложно, то есть `not zero(10)` истинно, тогда, упрощая `15 mod 10` до `5`, мы вычеркиваем `нод (15, 10, x)` из обеих частей и получаем подцель:

```
← нод(10, 5, x)
```

Заметьте, что это другая цель, которая будет разрешена как оригинальная цель. Здесь выполняем унификацию:

```
нод(10, 5, x) ← not zero(5), нод(5, 10 mod 5, x), нод(10, 5, x)
```

И таким образом получаем последующую подцель:

```
← нод(5, 0, x)
```

Теперь она сопоставляется с первым правилом:

```
нод(u, 0, u)
```

Следовательно, конкретизируя `x` значением `5`, приходим к пустому утверждению, а система отвечает:

```
Yes: x = 5
```

Резолюция и унификация вычислили наибольший общий делитель для `10` и `15`, используя алгоритм Евклида!

Теперь рассмотрим другую задачу, которая должна быть решена перед выбором рабочей реализации резолюции. Чтобы обеспечить эффективное исполнение, система логического программирования должна применять такой алгоритм, который определяет:

1) порядок, в котором система пытается разрешить список целей,
2) порядок, в котором формулы используются для разрешения целей.

Как пример списка целей, рассмотрим цель:

```
← ноги(лошадь, 4)
```

Напомним, что ее обработка привела к двум подцелям:

```
← млекопитающее(лошадь), руки(лошадь, 0)
```

Система логического программирования должна выбрать цель, которой она будет заниматься в первую очередь (`млекопитающее(лошадь)` или `руки(лошадь, 0)`). В этом примере любой выбор приведет к одинаковому ответу. Тем не менее, как мы увидим в дальнейшем, порядок разрешения целей может оказать существенное влияние на получаемый ответ.

Порядок, в котором используются формулы, также может повлиять на результат применения резолюции.

Например, для заданных формул Хорна:

```
прародитель (x, y) ← родитель(x, z), прародитель (z, y).
прародитель (x, x).
родитель (liza, tom).
```

если мы обеспечим запрос:

```
← прародитель (x, tom)
```

то возможны два ответа: `x = tom` и `x = liza`. Если используется утверждение `прародитель (x, x)`, будет найдено решение `x = tom`. Если используется формула `прародитель (x, y) ← родитель(x, z), прародитель (z, y)`, будет найдено решение `x = liza`. Все зависит от порядка использования формул, а также от порядка разрешения целей.

Реальные системы применяют формулы Хорна и резолюцию с предопределенным порядком относительно целей и использования формул, поэтому они нарушают основной принцип логического программирования: программисты должны беспокоиться только о логике, в то время как элементы управления (методы, используемые для создания ответа) должны игнорироваться. На самом же деле программист всегда должен знать методику, по которой система создает ответы. Конечно, возможно создание таких систем, которые будут генерировать всевозможные ответы, допускаемые логикой, но пока они слишком неэффективны, чтобы использоваться как системы программирования.

Язык Prolog

Prolog — наиболее известный язык логического программирования [4, 23, 31,32, 41, 74]. Он использует формулы Хорна и реализует резолюцию с поиском «сначала-вглубь» и алгоритмом унификации. В настоящее время существует стандарт ISO для языка Prolog, однако много лет не было никакого стандарта, поэтому отдельные реализации сильно отличались друг от друга по синтаксису, семантике, встроенным функциям и библиотекам. Наиболее распространенной реализацией считается эдинбургский Prolog (развивался в конце 70-х и в начале 80-х в университете Эдинбурга). Он стал фактическим стандартом и базисом для стандарта ISO. Мы используем нотацию эдинбургского языка Prolog во всех примерах.

Нотация и структуры данных

Prolog применяет нотацию почти идентичную той, которая рассматривалась для формул Хорна, за исключением одного обозначения: стрелка импликации ← за-

менена двоеточием, за которым следует черточка (или знак «минус») (:-). Таким образом, программа прародителя из предыдущего раздела была бы написана в синтаксисе языка Prolog следующим образом:

```
прародитель (X, Y) :- родитель (X, Z), прародитель (Z, Y).
прародитель (X, X).
родитель (liza, tom).
```

а пример о числах был бы написан в следующем виде:

```
natural(0).
natural(successor(X)) :- natural(X).
```

Обратите внимание, что названия переменных `X` и `Y` набраны в верхнем регистре. Prolog отличает переменные от констант и имен предикатов и функций за счет использования верхнего регистра для переменных и нижнего регистра для констант и имен. В некоторых системах Prolog при записи переменной вводят символ подчеркивания перед именем:

```
прародитель (_x, _x)
```

Кроме того, вместо соединителя `and` используют запятую, а вместо `or` — точку с запятой. Правда, точка с запятой редко применяется в программировании, так как она не является стандартной частью логики формулы Хорна.

Базовой структурой данных являются термы, например `родитель (X, Z)` или `successor(successor(0))`. Кроме того, в качестве базовой структуры данных часто применяют список. Список в языке Prolog заключается в квадратные скобки. Если список включает элементы `x`, `y`, и `z`, то это записывается как `[x, y, z]`. Списки могут содержать термы или переменные.

С помощью вертикальной черты можно определить голову и хвост списка. Например, `[H|T]` означает, что `H` — первый элемент в списке, а `T` является хвостом списка. Это тип нотации может использоваться для извлечения компонентов списка через сопоставление с образцом. Таким образом, если `[H|T] = [1,2,3]`, то `H = 1` и `T = [2,3]`. Перед чертой можно написать сколько угодно термов. Например, `[X, Y|Z] = [1,2,3]` дает `X = 1, Y = 2, a Z = [3]`. Пустой список обозначается как `[]` и у него нет первого элемента.

В языке Prolog есть несколько стандартных встроенных предикатов, например `not`, `=`, а также операции ввода-вывода: `read`, `write` и `nl` (для newline). Одна операция в языке Prolog обозначается необычно: меньше чем или равно записывается как `=<` вместо `<=` (наверное потому, что последняя комбинация уж слишком похожа на импликацию.

Выполнение в среде языка Prolog

Компиляторы для языка Prolog существуют, но большинство систем работает как интерпретаторы. Программа на Prolog состоит из набора формул Хорна в синтаксисе языка Prolog, которые обычно вводятся из файла и сохраняются в базе данных формул. После ввода набора формул в базу данных для начала выполнения могут вводиться цели. Цели вводятся из файла или с клавиатуры. Прежде всего система Prolog формирует для пользователя подсказку на запрос:

```
?-_
```

Обратите внимание, что синтаксис в Prolog теперь стандартизирован для формул и запросов, а вот реакция интерпретатора Prolog (например, приведенная нами подсказка) не стандартизирована; мы записываем эту реакцию в наиболее вероятной нотации, но различные интерпретаторы могут вести себя и по-другому.

Приведем иллюстрацию. Если формулы

```
прародитель(X, Y) :- родитель(X, Z), прародитель(Z, Y).
прародитель(X, X).
родитель (liza, tom).
```

были введены в базу данных, тогда следующие запросы вызовут обозначенную реакцию:

```
?- прародитель((liza, tom).
yes.
?- прародитель(tom, liza).
no.
?- прародитель (X, tom).
X = liza ->_
```

На последний запрос есть два ответа. Большинство систем для языка Prolog найдет один ответ и затем станет ждать реакции пользователя (для печати следующего ответа). Если пользователь вслед за символом подчеркивания на экране (его печатает система) введет точку с запятой (имеет значение `or`), тогда Prolog продолжит искать ответы:

```
?- прародитель(X,tom).
X = liza ->;
X = tom
?-_
```

Символ возврата каретки обычно аннулирует продолжение поиска.

Арифметика

У языка Prolog есть встроенные арифметические операции и арифметический вычислитель. Арифметические термы могут быть написаны или в обычной инфиксной нотации, или как термы в префиксной нотации: `3 + 4` и `+ (3, 4)` означают одно и то же.

Однако Prolog не может решить, рассматривать ли арифметический терм как терм непосредственно (то есть как данные) или следует вычислить его значение. Таким образом:

```
?- write(3 + 5).
3 + 5
```

Для вычисления значения арифметического терма требуется применить встроенную операцию — предикат `is`. Чтобы найти значение величины `3 + 5`, нужно написать:

```
?- X is 3 + 5, write(X).
X = 8
```

Следствием такого подхода является тот факт, что два арифметических терма могут считаться различными, даже если у них одно и то же значение:

```
?- 3 + 4 = 4 + 3 .
No
```

ПРИМЕЧАНИЕ

Мы пишем пробел между 3 и последующей точкой, так как 3. может интерпретироваться как число с плавающей точкой 3.0.

Для оценки равенства значений следует использовать предикат `is`, записав:

```
valequal(Term1, Term2) :-
X is Term1, Y is Term2, X = Y.
```

Теперь мы бы получили:

```
?- valequal(3 + 4, 4 + 3).
yes
```

Рассмотрим, как записать алгоритм Евклида для наибольшего общего делителя на языке Prolog. В терминах формул Хорна он имеет вид:

```
нод(u, 0, u).
нод(u, v, w) ← not zero(v), нод(v, u mod v, w).
```

Результат трансляции на Prolog принимает форму:

```
нод(U, 0, U).
нод(U, V, W) :-
    not(V = 0), R is U mod V, нод(V, R, W).
```

Здесь среднее утверждение `R is U mod V` задает вычисление операции `mod`.

Унификация

Унификация — это процесс, в котором для достижения совпадения с образцом (в ходе резолюции) выполняется конкретизация переменных, то есть их связывание со значениями. В результате двум термам придается одинаковый смысл. Базовым выражением, смысл которого определен унификацией, является выражение эквивалентности: в языке Prolog цель `s = t` пытается унифицировать термы `s` и `t`. Цель достигается (считается истиной), если унификация оказывается успешной, и терпит неудачу в противном случае. Таким образом, мы можем изучить унификацию в языке Prolog, экспериментируя с эффектом эквивалентности:

```
?- me = me.
yes
?- me = you.
no
?- me = X.
X = me
?- f(a, X) = f(Y, b).
X = b
Y = a
?- f(X) = g(X).
no
?- f(X) = f(a, b).
no
?- f(a, g(X)) = f(Y, b).
no
?- f(a, g(X)) = f(Y, g(b)).
X = b
Y = a
```

На основе экспериментов сформулируем следующие правила унификации для системы Prolog:

1. Константа унифицируется лишь сама с собой: `me = me` успех, но `me = you` отказ.
2. Неконкретизированная переменная унифицируется с чем угодно и становится конкретизированной.
3. Структурированный терм (например, функция, применяемая к аргументам) унифицируется с другим термом только тогда, когда у него такое же имя функции, такое же число аргументов и аргументы могут быть рекурсивно унифицированы. Например, `f(a, X)` унифицируется с `f(Y, b)`, с конкретизацией `b` для `X` и `a` для `Y`.

Вариация для второго правила, когда унифицируются две неконкретизированные переменные, имеет вид:

```
?- X = Y.
X = _23
Y = _23
```

Число `23` в данном случае ассоциируется с областью памяти, зарезервированной для этой переменной. Таким образом, унификация приводит к тому, что неконкретизированные переменные начинают совместно использовать память (то есть становятся алиасами друг для друга).

Унификация в Prolog позволяет получить очень короткие выражения для многих операций.

Создадим пару Prolog-программ для операций `append` и `reverse` над списками. Поскольку в языке Prolog список можно представлять термом, таким как `[X|Y]`, где `X` и `Y` — переменные, здесь нет нужды возвращать голову или хвост списка с помощью встроенной функции. Достаточно установить `[X|Y] = [1, 2, 3]`, и неконкретизированные переменные `X` и `Y` вернут заголовок как `X`, а хвост как `Y` за счет унификации. Если нужно добавить к списку `[ 1, 2, 3 ]`, скажем, `0`, мы просто пишем `[ 0 | [1,2,3] ]`; например:

```
?- X = [0|[1, 2, 3]].
X = [0, 1, 2, 3]
```

Таким образом, мы можем использовать переменные в терме как входные или как выходные параметры, а формулы языка Prolog могут «запускаться» как назад, так и вперед, хотя процедурная интерпретация формул Хорна не говорила нам об этом! Действительно, унификация может использоваться, чтобы сократить выражения. Начиная с того места, где стоит операция `=`, мы позволяем резолюции вызывать унификацию автоматически, записывая образцы, прямо унифицируемые в параметрах головы. Например, можно записать:

```
cons(X, Y, [X|Y])
```

Появление образца [X|Y] вместо третьей переменной автоматически унифицирует его с переменной, используемой в этом месте цели. Этот процесс можно назвать как вызов, направленный на образец.

Теперь запишем процедуру `append`:

```
append(X, Y, Z) :- X = [], Y = Z.
append(X,Y,Z) :-
    X = [A|B], Z = [A|W], append(B, Y, W).
```

Первая формула заявляет, что добавление любого списка к пустому списку просто воспроизводит тот же список. Вторая формула утверждает, что добавление списка с головой A и хвостом B к списку Y дает список, головой которого также является A, а хвост — это B с добавленным Y.

Перезаписывая это с помощью вызова, направленного на образец, получаем чрезвычайно краткую форму:

```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).
```

Подобное добавление может выполняться и с другой стороны. При формировании конкретного списка оно способно обнаружить все способы добавления для двух списков:

```
?- append(X, Y, [1, 2]).
X = []
Y = [1, 2] ->;

X = [1]
Y = [2] ->;

X = [1,2].
Y = []
```

Prolog использует первую формулу и сопоставляет X пустому списку [], а Y — конечному списку. Затем он продолжает поиск решений с помощью второй формулы, сопоставляя X с [A|B] и устанавливая подцель append(B, Y, W) при W = [2]. Далее поиск начинается снова с B вместо X. Таким образом, B сначала сопоставляется с [] а Y с [2], формируя X = [1 | []] = [1]. Затем B сопоставляется с новым [A|B] по второй формуле и т. д.

В заключение дадим определение в языке Prolog для реверса списка:

```
reverse([],[]).
reverse([H|T], L) :- reverse(T, L1),
    append(L1, [H], L).
```

Стратегия поиска в языке Prolog

Prolog проводит резолюцию в строго линейной стиле, заменяя цели слева направо и рассматривая формулы в базе данных в порядке сверху-вниз. Подцели также рассматриваются немедленно, по мере их установки. Таким образом, эта стратегия поиска реализуется как поиск в глубину на дереве возможных вариантов.

Например, рассмотрим следующие формулы:

```
(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(liza, tom).
```

Для цели ancestor(X, tom) стратегия поиска Prolog (слева направо и сначала в глубину) на дереве подцелей показана на рис. 21.1. Ребра на этом рисунке помечены номерами формул, используемых системой Prolog для резолюции, а конкретизации переменных записаны в фигурных скобках.

Листовые вершины в этом дереве достигаются, если не найдено совпадение для крайней левой формулы или если все формулы были устранены, что соот-

ветствует успеху. Всякий раз, когда зафиксирован отказ (или пользователь задал непрерывный поиск с помощью точки с запятой), Prolog выполняет откат вверх по дереву, чтобы обнаружить другие пути к листу, освобождая конкретизации переменных. Таким образом, если после нахождения в дереве решения `X = liza` инициируется откат, эта конкретизация `X` будет освобождена и будет найден новый путь к листу с `X = tom`.

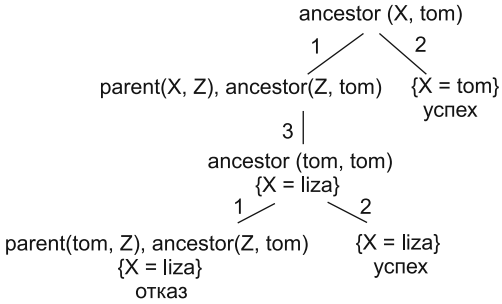


Рис. 21.1. Дерево поиска для Prolog, показывающее подцели, формулы, используемые для резолюции, и конкретизации переменных

Эта стратегия «сначала в глубину» чрезвычайно эффективна, поскольку она может быть реализована на основе стека или в рекурсивном стиле. Однако она же означает, что решения не могут быть найдены, если в дереве поиска имеются ветви бесконечной глубины. Например, предположим, что мы написали формулы в несколько отличном порядке:

```
(1) ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).
(2) ancestor(X, X).
(3) parent(liza, tom).
```

Такая запись заставит Prolog войти в бесконечный цикл в попытках удовлетворить `ancestor(Z, Y)` путем многократного использования первой формулы. Конечно, это является следствием леворекурсивного способа, которым была написана первая формула, и того факта, что никакие другие формулы не предшествуют ей. В «чистом» логическом программировании, где используется поиск «вначале в ширину» («сначала вширь»), всегда будут искаться и другие решения (если они есть). К сожалению, поиск типа «сначала вширь» значительно дороже поиска в глубину, поэтому лишь немногие системы логического программирования используют его. Prolog всегда использует поиск в глубину.

Циклы и структуры управления

Мы можем использовать поиск в глубину с откатами системы Prolog для выполнения циклов и периодического повторения поиска. Для этого надо организовать откат даже в случае, когда решение найдено. Такую возможность предоставляет встроенный предикат `fail`. Например, мы можем поручить Prolog напечатать все решения для цели `append`, без ввода точки с запятой в систему. Определим предикат:

```
printpieces(L) :-append(X, Y, L),
    write(X),
    write(Y),
    nl,
    fail.
```

Теперь мы получаем следующее поведение:

```
?- printpieces([1, 2]).
[][1,2]
[1][2]
[1,2] []
no
```

Откат при отказе вынуждает Prolog написать все решения сразу. Эту методику можно применить и для получения периодически повторяющихся вычислений. Например, следующие формулы генерируют все целые числа, для которых $X \geqslant 0$, как решения цели `num(X)`:

```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

Дерево поиска отображено на рис. 21.2. Правая ветвь этого дерева имеет бесконечную длину (различные использования `Y` из выражения (2) в дереве обозначены добавлением кавычек к имени переменной).

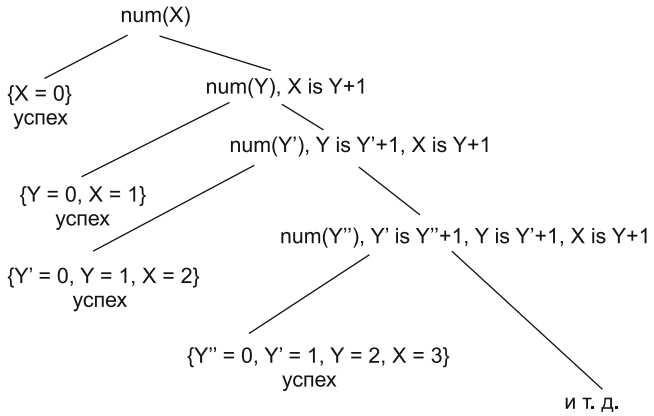


Рис. 21.2. Бесконечное дерево поиска для Prolog, показывающее периодически повторяющиеся вычисления

Теперь мы могли попытаться генерировать целые числа, скажем, от 1 до 10, записывая:

```
writenum(l, J) :- num(X),
    I =< X,
    X =< J,
    write(X),
    nl,
    fail.
```

и предоставляя цель `writenum (1,10)`. К сожалению, после `X = 10` система войдет в бесконечный цикл, генерируя даже большие целые для `X`, хотя эти значения никогда не потребуются.

Понятно, что необходимо остановить продолжение поиска по дереву. Для этого в языке Prolog имеется оператор отсечения, обозначаемый восклицательным знаком. При выполнении *оператор отсечения* «замораживает» выбор. Если отсечение достигнуто при откате, поиск в поддеревьях, родительская вершина которых содержит отсечение, останавливается. Поиск продолжается с вершины прародителя. В действительности отсечение сокращает дерево поиска, ограничивая его братьями (вершинами такого же уровня), расположенными справа от вершины, содержащей отсечение.

Рассмотрим дерево на рис. 21.1. Если мы введем отсечение:

```
(1) ancestor(X, Y):- parent(X, Z), !, ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(liza, tom).
```

тогда только решение `x = liza` будет найдено, так как ветвь, содержащая `X = tom`, будет удалена из поиска, как показано на рис. 21.3.

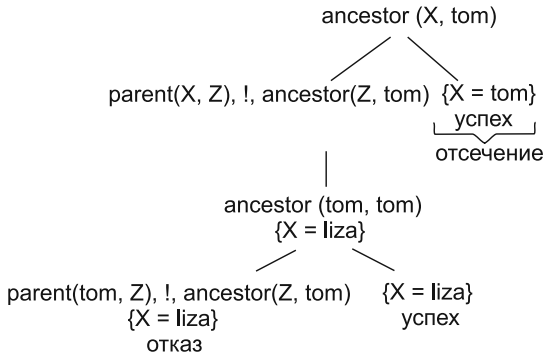


Рис. 21.3. Следствия отсечения для дерева поиска из рис. 21.1

С другой стороны, если мы размещаем отсечение следующим образом:

```
(1) ancestor(X, Y) :- !, parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(liza, tom).
```

тогда никакие решения вообще не будут найдены, как показано на рис. 21.4.

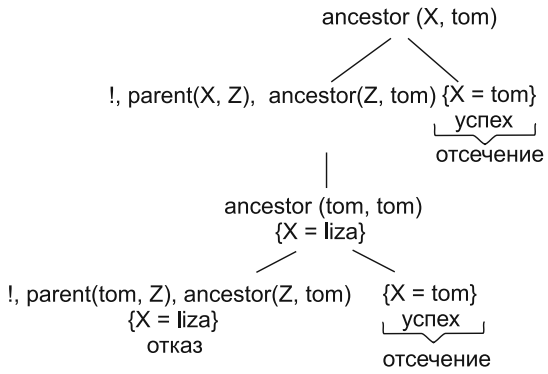


Рис. 21.4. Такое использование отсечения сокращает все решения из рис. 21.1

Наконец, если мы изменим размещение отсечения:

```
(1) ancestor(X, Y) : - parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X) : - !.
(3) parent(liza, tom).
```

оба решения будут все еще находиться, так как правое поддерево для `ancestor(X, tom)` не отсечено (фактически в этом примере ничего не сокращено).

Отсечение может использоваться как эффективный механизм для сокращения количества ветвей в текущем дереве поиска. Отсечение позволяет решить проблему бесконечного цикла в программе для печати чисел между I и J. Рассмотрим такое решение:

```
num(0).
num(X) : - num(Y), X is Y + 1.
writenum(I, J) :- num(X),
    I =< X,
    X =< J,
    write(X), nl,
    X = J, !,
    fail.
```

В этом коде цель `X = J` будет достигнута, когда будет достигнута верхняя граница `J`, а затем отсечение вызовет откат по `fail`, останавливая поиск новых значений `X`.

Отсечение может также использоваться для имитации конструкции `if-else` из императивных и функциональных языков. Чтобы написать такое выражение:

```
D = if A then B else C
```

создадим следующую Prolog-программу:

```
D :- A, !, B.
D :- C.
```

Обратите внимание, что мы можем достичь почти такого же результата без отсечения:

```
D :- A, B.
D :- not(A), C.
```

но здесь имеется некоторое отличие, так как `A` выполняется дважды. Конечно, если у `A` нет никаких побочных эффектов, то две формы эквивалентны. Однако отсечение действительно улучшает эффективность.

Использование оператора отсечения иногда сравнивается с применением оператора безусловного перехода в императивных языках. Несмотря на то что зачастую он действительно нужен, его можно критиковать. Действительно, иногда оператор отсечения вносит в логические программы поток управления, свойственный стилю императивного программирования.

Возможность вмешиваться в поток управления для программ на языке Prolog считается недостатком, поскольку это явно противоречит важнейшему преимуществу логического программирования, состоящему в том, что программы не определяют способ, которым должно быть найдено решение. Они просто указывают, как должно выглядеть решение. Подобный подход упрощает создание программ. Они не загромождены описаниями алгоритма решения, не задают точный порядок, в котором должны выполняться вычисления для получения решения. Таким обра-

зом, в то время как логическое программирование не требует определения потока управления в программе, язык Prolog часто вводит такой поток, в основном для повышения эффективности выполнения программ.

Проблемы языка Prolog

При использовании Prolog в качестве языка логического программирования возникает ряд проблем. Несмотря на то что этот язык является самым распространенным и полезным инструментом, его нельзя назвать ни «чистым», ни совершенным языком логического программирования.

Проблема проверки вхождения при унификации

Алгоритм унификации, используемый языком Prolog, фактически неверен. Унифицируя переменную `X` термом `T`, Prolog не проверяет, не входит ли сама переменная в терм, которым она конкретизируется. Это называют проблемой проверки вхождения. Когда `X` действительно входит в `T`, тогда унификация `X` и `T` может привести к бесконечным вычислениям. Например, рассмотрим

```
?- append([], E, [a, b|E]).
E = [a, b, a, b a, b, a, b, a, b, a, b, …
```

Для `append([], E, [a, b|E])` при унификации с `append([], Y, Y)` переменная `Y` должна унифицироваться как с `E`, так и с термом `[a, b|E]`, содержащим `E`.

Prolog пренебрегает проверкой вхождения `E` в `[a, b|E]`. При подстановке `[a, b|E]` в `E` получаем:

```
E = [a, b|E] = [a, b, a, b|E] = [a, b|E] = [a, b, a, b, a, b|E] = …
```

Проверка вхождения служит для того, чтобы предотвратить возникновение бессмысленных конкретизаций (присваиваний) вида `X := f(X)`.

Почему унификация в Prolog не содержит проверку таких вхождений? Дело в том, что унификация без проверки вхождения легко реализуется и требует малых затрат времени, в то время как эффективные алгоритмы, которые включают проверку вхождения, являются более сложными.

Проблема замкнутого мира

Природа резолюции в Prolog иногда приводит к ошибочным результатам. Истиной здесь считается только то, что может быть доказано с использованием базы данных Prolog. Система не имеет никаких других сведений о мире, кроме знаний, хранящихся в ее базе данных. Любой запрос считается ложным, если в базе данных недостаточно информации для его доказательства. Система Prolog может доказать, что данная цель является истинной, но она не может утверждать, что данная цель является ложной. Она просто предполагает, что эта цель должна быть ложной. По существу, решения системы Prolog относятся к категориям «истина или отказ», а не к категориям «истина или ложь».

ПРИМЕЧАНИЕ

Можно считать, что система Prolog обучалась на юридическом факультете — вспомните о презумпции невиновности.

Проблема логического отрицания

Другая проблема языка Prolog связана с логическим отрицанием. Положим, что имеются два факта и одно правило:

```
parent(alex, peter).
parent(alex, paul).
sibling(X, У) :- (parent(F, X), parent(F, У)).
```

Будем считать, что мы ввели запрос

```
?- sibling(X, У)•
```

Система ответит

```
X = peter
У = peter
```

Таким образом, система считает, что Петр является братом (`sibling`) самому себе. Это происходит потому, что система сначала конкретизирует переменную `F` значением `alex`, а переменную `X` — значением `peter` для того, чтобы сделать первую подцель `parent(F, X)` истинной. Затем она снова начинает поиск с начала базы данных для того, чтобы сопоставить вторую подцель `parent(F, У)`, и обнаруживает совпадение, когда переменная `F` конкретизирована значением `alex`, а переменная `У` — значением `peter`. Поскольку две подцели удовлетворяются независимо друг от друга, причем оба совпадения находятся в начале базы данных, возникает ответ `peter`. Для того чтобы избежать этого, следует указать, что переменная `X` находится в отношении `sibling` с переменной `У`, только если обе они находятся в отношении `parent` с одной и той же переменной и не совпадают друг с другом. К сожалению, непосредственно указать, что они не равны между собой, в языке Prolog нельзя. Наиболее точный метод может потребовать добавления в базу данных фактов для каждой пары термов о том, что они не совпадают между собой. Это приведет к тому, что база данных станет очень большой, поскольку негативной информации намного больше, чем позитивной. Например, у большинства людей количество дней, не являющихся их днями рождения, на 364 больше, чем количество их дней рождения.

Простая альтернатива — указать в цели, что переменная `X` не должна совпадать с переменной `У`:

```
sibling (X, У) :- parent(F, X), parent(F, У), not(X =У).
```

Будем считать, что нам повезло — решение оказалось простым.

Предикат `not` в языке Prolog удовлетворяется, если резолюция не может удовлетворить подцель `X = У`. Следовательно, если предикат `not` достигается, это не обязательно означает, что переменная `X` не равна переменной `У`. Скорее это означает, что резолюция не может вывести из базы данных, что переменная `X` равна переменной `У`. Таким образом, предикат `not` в языке Prolog отличается от логической операции отрицания, в которой отрицание означает, что ее операнд имеет истинное

значение. Эта неэквивалентность может привести к проблеме, если мы имеем дело с целью в следующем виде:

```
not(not(цель))
```

Если бы предикат `not` в языке Prolog был истинной операцией логического отрицания, наша цель могла бы быть эквивалентна цели:

```
цель
```

В некоторых случаях, увы, они не совпадают.

Важнейшее свойство логических программ — «добавление информации к системе может уменьшить количество утверждений, которые могут быть доказаны» — называют *немонотонным выводом*. Оно является следствием проблемы замкнутого мира. Немонотонный вывод пристально изучается в логическом программировании и теории искусственного интеллекта.

Связанная проблема состоит в том, что отказ заставляет отменять конкретизации переменных при откате. Таким образом, после отказа у переменной может и не быть соответствующего значения. Отсюда следует, например, что `not(not(X))` не имеет того же значения, что и `X` (в этом примере мы подразумеваем факт `human(bob)`):

```
?- human(X).
X = bob
?- not(not(human(X))).
X = _23
```

Цель `not(not(human(X)))` достигается потому, что отказывает `not(human(X))`, но когда `not(human(X))` отказывает, конкретизация `X` значением `bob` освобождается, вследствие чего `X` распечатывается как неконкретизированная переменная.

Подобная ситуация показана в следующем фрагменте:

```
?- X = 0, not(X = 1).
X = 0
?- not (X = 1), X = 0.
no
```

Вторая пара целей отказывает, потому что для достижения цели `X = 1` переменная `X` конкретизирована единицей, а затем для `not (X = 1)` фиксируется отказ. Цель `X = 0` не достигается никогда.

Формулы Хорна не выражают всю логику утверждений

Не каждое логическое утверждение может быть превращено в формулу Хорна. В частности, использование в утверждениях кванторов не может быть выражено в форме формулы Хорна. Приведем простой пример:

```
p(a) and (there exists x, not(p(x)))
```

Мы можем, конечно, выразить истинность `p(a)` в виде формулы Хорна, но второе утверждение не может быть записано как формула Хорна. Если мы попытаемся следовать стилю языка Prolog, то получится следующее:

```
p(a).
not(p(b)).
```

Правда, второе утверждение инициирует ошибку (при попытках переопределить логическую операцию `not`). Лучшим решением, основанным на предположении о проблеме замкнутого мира, может быть утверждение `not(p(X))`, которое должно быть истинным для всех значений `X`, не эквивалентных `a`. Но реально здесь воспроизводится вот такое утверждение:

```
p(a) and (for all x, not(x = a) → not(p(a)))
```

А оно отличается от исходного утверждения.

Информация управления в логическом программировании

Мы уже обсуждали отсечение как полезный механизм явного управления в языке Prolog. Кроме этого, принятая в системе Prolog стратегия поиска в глубину, а также линейная обработка целей и утверждений вносят в программы неявную информацию об управлении, которая может стать причиной отказа. Рассмотрим пример:

```
ancestor(X, Y) : - parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(liza, tom).
```

Если бы мы случайно перепутали порядок элементов в правой части первого утверждения:

```
ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z)
```

мы немедленно вошли бы в бесконечный цикл: для обеспечения истинности цели `ancestor` Prolog попытается конкретизировать переменную `Z`, выполняя бесконечный спуск в дереве поиска на основе первой формулы. С другой стороны, если мы изменим порядок формул:

```
ancestor(X, X).
ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).
parent(liza, tom).
```

поиск обнаружит оба решения `X = liza` и `X = tom` для запроса `ancestor(liza, X)`, но снова войдет в бесконечный цикл, отыскивая дальнейшие несуществующие решения. К аналогичной ситуации приведет реверс порядка формул в программе для генерации натуральных чисел:

```
num(X) : - num(Y), X is Y + 1 .
num(0).
```

Теперь цель `num(X)` приведет к бесконечному циклу и не будет сгенерировано ни одно число.

Более сложный логический вопрос — это представление с помощью формул Хорна логики алгоритмов.

Приведем спецификацию процедуры сортировки для двух списков `S` и `T`:

```
sort(S, T) :- permutation(S, T), sorted(T)
```

Запишем спецификации для перестановки и сортировки в синтаксисе языка Prolog:

```
sorted([]).
sorted([X]).
```

```
sorted([X, Y|Z]) :- X =< Y, sorted([Y|Z]).
permutation([], []).
permutation(X, [Y|Z]) :- append(U, [Y|V], X),
    append(U, V, W),
    permutation(W, Z).
```

Фактически здесь представлено математическое определение того, как список чисел будет отсортирован в порядке увеличения значений. Но эта программа реализует самую медленную сортировку в мире. Перестановки не отсортированного списка генерируются до тех пор, пока одна из них (случайно) не окажется отсортированной!

В лучшем из миров хотелось бы, чтобы система логического программирования обработала математическое определение и нашла эффективный алгоритм для его вычисления. Конечно, в системе Prolog (или в любой другой системе логического программирования) нужно не только предоставлять спецификации программ, но и обеспечивать информацию об алгоритме управления. В частности, для программы НОД мы явно определили шаги, используемые алгоритмом Евклида, вместо записи математического определения наибольшего общего делителя. Точно так же для получения программы эффективной сортировки в Prolog следует определить фактические шаги реализующего алгоритма. Например, Prolog-программа быстрой сортировки `quicksort` может иметь следующий вид:

```
qsort([], []).
qsort([H|T], S) :- partition(H, T, L, R),
    qsort(L, L1),
    qsort(R, R1),
    append(L1, [H|R1], S).
partition(P, [A|X], [A|Y], Z) :- A < P,
    partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A >= P,
    partition(P, X, Y, Z).
partition(P, [], [], []).
```

В этой программе стандартный метод разбиения представлен формулами для предиката разбиения, а два рекурсивных вызова `qsort` (во втором утверждении) вполне очевидны. Таким образом, при спецификации последовательных алгоритмов (например, сортировки) Prolog не слишком отличается от императивных или функциональных языков программирования, как можно было подумать.

Конечно, очень важно сформировать качественную спецификацию, даже если она не задает алгоритм вычислений. Не менее важно тестировать правильность спецификации. Prolog может с успехом использоваться для практической проверки спецификаций (во время проектирования программ).

Контрольные вопросы и упражнения

1. Запишите следующие высказывания в терминах исчисления высказывания:
   - Если идет дождь или снег, то это значит, что есть осадки.
   - Если сейчас зима и есть осадки, то это значит, что идет снег.
   - Если сейчас не зима и есть осадки, то это значит, что идет дождь.
   - Идет снег.

2. Запишите высказывания из предыдущего упражнения в терминах формул языка Prolog (в указанном порядке). Какой ответ даст Prolog на вопросы «Это зима?», «Идет дождь?» Почему? Можете ли вы переупорядочить высказывания так, чтобы Prolog мог давать лучшие ответы?
3. Запишите следующее высказывание в виде формул Prolog: «млекопитающие имеют четыре ноги и не имеют рук, или две руки и две ноги».
4. Добавьте высказывание «лошадь — это млекопитающее, и лошадь не имеет рук» к формулам из предыдущего упражнения. Может ли Prolog вывести, что у лошади четыре ноги? Ответ поясните.
5. Запишите формулы Prolog для выражения следующих отношений: при данном родителе описать отношения с дедушкой, бабушкой, братом, сестрой, двоюродным братом.
6. Запишите программу на языке Prolog для нахождения последнего элемента в списке.
7. Запишите программу на языке Prolog для нахождения минимального и максимального значений в списке из целых чисел.
8. Язык Prolog разделяет некоторые черты функциональных языков Scheme и ML, рассмотренных в главе 20. Опишите две главные сходные черты. Опишите два главных различия.
9. Положим, что для заданной формулы

```
human(bob)
```

   система Prolog формирует следующую последовательность:

```
?- human(X).
X = bob
?- not(human(X)).
no
```

   Почему система Prolog так отреагировала на последнюю цель? Утверждает ли Prolog, что если нет X, которые не «human», то это значит, что все X являются «human»? Почему?

Заключение

— Заклинаю тебя, о юный мой читатель, — горделиво обратился автор, нарушив довольно продолжительное молчание, — потряс ли ты своими знаниями учителей своих и товарищей своих?

— Потряс! — вздохнул читатель и с ненавистью посмотрел на автора.

По мотивам «Старика Хоттабыча»

Любую работу трудно начинать, но еще трудней заканчивать.

Подошел к финалу учебник, посвященный языкам программирования — очень важному инструменту, обеспечивающему создание как маленьких, так и очень больших программных приложений.

Конечно, каждый уважающий себя специалист должен знать, как выглядит его рабочий инструмент не только извне, но и изнутри. Формированию этого внутреннего знания и была посвящена данная работа.

Можно сказать, что предложенный материал обеспечивал изучение и скрупулезный анализ не только внешнего, но и внутреннего мира языков программирования. Действительно, ведь здесь рассматривалось как «кроятся и шьются» языки программирования.

Начали мы с определения того, что такое синтаксис и что такое семантика языка программирования. Далее изучили основной инструмент определения синтаксиса — грамматику. Детально разобрали работу средства для проверки синтаксиса — дерева разбора. Затем переключились на обсуждение аппарата формального определения семантики: атрибутной, операционной, аксиоматической и денотационной семантики.

Отметив, что любой язык включает в себя средства описания действий и данных, мы подробно описали выражения, операторы, способы управления последовательностью действий, после чего обратились к типам данных, их экземплярам (объектам)

для наиболее популярных ЯП, не оставив в стороне проблематику взаимодействия программ с внешней средой (иначе говоря, проблематику ввода-вывода), а также проблематику исключений — аварийных сигналов, возникающих в форс-мажорных обстоятельствах.

Говоря о полном жизненном цикле объектов данных, мы изучили их характеристики, связывания различных характеристик в ходе выполнения программы, способы преобразования и проверки типов данных. Были подробно рассмотрены разнообразные вопросы создания и использования «рукотворных» операторов и операций — методы создания и использования подпрограмм. Здесь же были изучены методы управления подпрограммами в период выполнения программы.

Перейдя к положениям «крупноблочного» строительства программ, мы обсудили средства для поддержки абстракций данных — абстрактные типы данных и классы, детально разобрав достоинства механизма инкапсуляции. Здесь же мы отметили возможности надстройки к объектно-ориентированному подходу, реализуемой аспектами, которые комфортно функционируют в межклассовом пространстве.

Покинув область традиционных последовательных вычислений, мы вторглись в раздел параллельного программирования, обеспечивающего организацию параллельных вычислений. Далее, не останавливаясь на достигнутом, изложение было сосредоточено на идеях и базовых понятиях функционального и логического подходов к программированию.

Все это вы теперь знаете, уважаемый читатель. Пусть эти знания сослужат вам добрую службу в нелегком, но очень увлекательном занятии, которым является программирование! В добрый путь, практикующие программисты!

Список литературы

1. *Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д.* Компиляторы. Принципы, технологии, инструментарий, 2-е изд. — М.: Издательский дом «Вильямс», 2008. — 1184 с.
2. *Бадд Т.* Объектно-ориентированное программирование в действии. — СПб.: Питер, 1997. — 464 с.
3. *Бен-Ари М.* Языки программирования. Практический сравнительный анализ. — М.: Мир, 2000. — 366 с.
4. *Братко И.* Алгоритмы искусственного интеллекта на языке PROLOG. 3-е изд. — М.: Издательский дом «Вильямс», 2004. — 640 с.
5. *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. — СПб.: Символ-Плюс, 1999. — 304 с.
6. *Глушков В. М.* Теория автоматов и формальные преобразования микропрограмм. — Кибернетика, 1965, № 5. С. 1–10.
7. *Валиков А. Н.* Технология XSLT. — СПб.: БХВ-Петербург, 2002. — 544 с.
8. *Вирт Н.* Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — 406 с.
9. *Вирт Н.* Систематическое программирование. Введение. — М.: Мир, 1977. — 184 с.
10. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. — М.: Мир, 1975. — 248 с.
11. *Дейкстра Э.* Дисциплина программирования. — М.: Мир, 1978. — 277 с.
12. *Калинин А. Г., Мацкевич И. В.* Универсальные языки программирования. Семантический подход. — М.: Радио и связь, 1991. — 400 с.
13. *Кауфман В. Ш.* Языки программирования. Концепции и принципы. — М.: Радио и связь, 1993. — 432 с.
14. *Керниган Б. В., Ричи Д. М.* Язык программирования С. 2-е изд. М.: Издательский дом «Вильямс», 2009. — 304 с.

15. *Клоксин У., Меллиш К.* Программирование на языке ПРОЛОГ. М.: Мир, 1987. — 336 с.

16. *Кубенский А. А.* Функциональное программирование. — СПб.: СПбГУ ИТМО, 2010. — 251 с.

17. *Лавров С. С.* Программирование. Математические основы, средства, теория. — СПб.: БХВ-Петербург, 2001. — 320 с.

18. *Лафоре Р.* Объектно-ориентированное программирование в C++. СПб.: Питер, 2003. — 928 с.

19. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования. М.: Мир, 1982. — 406 с.

20. *Липпман С., Лажойе Ж., Му Б.* Язык программирования C++. Вводный курс, 4-е изд. М.: Издательский дом «Вильямс», 2007. — 896 с.

21. *Лоусон Б., Шарп Р.* Изучаем HTML5. Библиотека специалиста. — СПб.: Питер, 2011. — 272 с.

22. *Лутц М.* Изучаем Python, 4-е изд. — СПб.: Символ-Плюс, 2011. — 1280 с.

23. *Малпас Дж.* Реляционный язык ПРОЛОГ и его применение. М.: Наука. Гл. ред. физ.-мат. лит., 1990. — 464 с.

24. *Одинцов И. О.* Профессиональное программирование. Системный подход. — СПб.: БХВ-Петербург, 2002. — 512 с.

25. *Орлов С. А.* Принципы языков программирования: Опорный конспект лекций. 5-е изд., дополнен. — Рига: TSI, 2008. — 108 с.

26. *Орлов С. А.* Принципы объектно-ориентированного и параллельного программирования на языке Ада 95.— Рига: TSI, 2001. — 327 с.

27. *Орлов С. А.* Принципы построения языков программирования. Учебник. — Рига: TSI, 2012. — 362 с.

28. *Орлов С. А., Цилькер Б. Я.* Организация ЭВМ и систем. Учебник для вузов. 2-е изд. СПб.: Питер, 2010. — 688 с.

29. *Орлов С. А., Цилькер Б. Я.* Технологии разработки программного обеспечения: Современный курс по программной инженерии. Учебник для вузов. 4-е изд. Стандарт третьего поколения. — СПб.: Питер, 2012. — 608 с.

30. *Подбельский В. В.* Язык C++. 5-е изд. М.: Финансы и статистика, 2003.— 560 с.

31. *Пратт Т., Зелковиц М.* Языки программирования: разработка и реализация, 4-е изд. — СПб.: Питер, 2002. — 688 с.

32. *Себеста Р. У.* Основные концепции языков программирования, 5-е изд. — М.: Издательский дом «Вильямс», 2001. — 672 с.

33. Семантика языков программирования. Под ред. В. М. Курочкина. — М.: Мир, 1980. — 397 с.

34. *Страуструп Б.* Программирование: принципы и практика использования C++. — М.: Издательский дом «Вильямс», 2011. — 1248 с.

35. *Тидуэлл Д.* XSLT. 2-е изд. — СПб.: Символ-Плюс; 2009. — 960 с.

36. *Филд А., Харрисон П.* Функциональное программирование: Пер. с англ. — М.: Мир, 1993. — 637 с.

37. *Флэнаган Д., Мацумото Ю.* Язык программирования Ruby. — СПб.: Питер, 2011. — 496 с.

38. *Хейлсберг А., Торгерсен М., Вилтамут С., Голд П.* Язык программирования C#. СПб.: Питер, 2012. — 784 с.

39. *Хоар Ч.* Взаимодействующие последовательные процессы. — М.: Мир, 1989. — 264 с.

40. *Хоган Б.* HTML5 и CSS3. Веб-разработка по стандартам нового поколения. — СПб.: Питер, 2012. — 272 с.

41. *Хоггер К.* Введение в логическое программирование. — М.: Мир, 1988. — 348 с.

42. *Чарнецки К., Айзенекер У.* Порождающее программирование: методы, инструменты, применение. Для профессионалов. — СПб.: Питер, 2005. — 731 с.

43. *Шварц Р., Феникс Т., Брайан Д. Фой.* Изучаем Perl, 5-е изд. англ. — СПб: Символ-Плюс, 2009. — 384 с.

44. *Эккель Б.* Философия Java. 4-е изд. — СПб.: Питер, 2009. — 640 с.

45. *Эндрюс Г. Р.* Основы многопоточного, параллельного и распределенного программирования. — М.: Издательский дом «Вильямс», 2003. — 512 с.

46. Языки программирования. Под ред. Ф. Женюи. — М.: Мир, 1972. — 408 с.

47. ACM Computing Surveys: Special Issue on Programming Language Paradigms (21)3 (1989).

48. ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)].

49. Aho, A., Sethi R., and Ullman J. D., Compilers: Principles, Techniques and Tools. — Addison-Wesley, Reading, MA (1988).

50. Backus, J., The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, Information Processing, UNESCO, Paris (1960), 125–132.

51. Backus, J., Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Comm. ACM (21)8 (1978), 613–641.

52. Barnes, J. Programming In Ada 2005. — Addison-Wesley, 2006. — 828 pp.

53. Böhm, C., and Jacobini G. J, Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules, Comm. ACM (9)5 (1966), 366–371.

54. Burns, A. Wellings, A. Concurrent and Real Time Programming in Ada — Cambridge University Press, 2007. — 461 pp.

55. Cardelli, L., Compiling a Functional Language, Symp. on LISP and Functional Programming, ACM (1984), 208–217.

56. Cardelli, L., Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. In ACM Computing Surveys, Vol. 17, No. 4, December 1985, pp. 471–522.

57. Chomsky, N., On Certain Formal Properties of Grammars, Information and Control 2 (1959), 137–167.

58. Dahl O., Dijkstra E., and Hoare C. A. R., Structured Programming. — Academic Press, New York (1972).

59. Deitel P. J., Deitel H. M. Internet & World Wide Web. How to program, 4th ed. — Pearson/Prentice-Hall, 2008. — 1407 pp.

60. Dijkstra, E. W. Goto Statement Considered Harmful, Commun. ACM, Vol. 11, No. 3, (1968), 147–149.

61. Dijkstra, E. W., Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, Comm. ACM (18)8 (1975), 453–457.

62. Feldman, M.B. Koffman, E.B. Ada 95: Problem Solving and Program Design. 2nd ed. — Addison-Wesley, 1996. — 814 pp.

63. Goldberg, A., Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA (1984).

64. Graham, P., On Lisp: Advanced Techniques for Common Lisp. — Prentice-Hall, Englewood Cliffs, NJ (1994).

65. Guttag, J. V., Notes on Type Abstraction (Version 2), IEEE Trans. on Software Engineering (6)1 (1980), 13–23.

66. Hoare, C. A. R. An axiomatic basis of computer programming. Communications of the ACM, 12(10):576–580, October 1969.

67. Hoare, C. A. R., and Wirth N., An Axiomatic Definition of the Programming Language Pascal, Acta Informatica 2 (1973), 335–355.

68. Hoare, C. A. R., and Lauer P. E., Consistent and Complementary Formal Theories of the Semantics of Programming Languages, Acta Informatica 3 (1974), 135–153.

69. Ierusalimschy, R. Programming in Lua, 2nd ed. — Rio de Janeiro, Lua.org, 2006. — 324 pp.

70. Knuth, D. E., Semantics of Context-Free Languages, Mathematical Systems Theory 2 (1968), 127–145.

71. Knuth, D. E. Structured Programming with GOTO Statements, ACM Computing Surveys, Vol. 6, No. 4, (1974), 261–301.

72. Laddad, R. AspectJ in Action, 2nd ed. — Manning, 2010. — 536 pp.

73. Linger, R. C., Mills H. D., and Witt B. I., Structured Programming: Theory and Practice, Addison-Wesley, Reading, MA (1979).

74. Louden, K. C., Lambert, K. A. Programming Languages: Principles and Practice, 3rd ed. — U.S.A. Course Technology Cengage Learning, 2011. — 671 pp.

75. McCarthy, J., LISP 1.5 Programmer's Manual, 2nd ed. — MIT Press, Cambridge, MA (1961).

76. Meyer, B., Eiffel: The Language. — Prentice-Hall, Upper Saddle River, NJ (1990).

77. Ousterhout J. Scripting: higher-level programming for the 21st century J/ IEEE Computer. — 1998. — 31(3), 23–30.

78. Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM (15)12 (1972), 1053–1058.

79. Ritchie, D. M., The Development of the C Language, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 201–208.

80. Pierce, B. C. Types and Programming Languages. — MIT Press, Cambridge, MA, 2002.

81. Pratt, T. W., and Zelkowitz M. V. Programming Languages: Design and Implementation, 4th ed. — Prentice Hall, Englewood Cliffs, 2001. — 649 pp.

82. Sammet, J., Programming Languages: History and Future, Comm. ACM (15)7 (1972), 601–610.

83. Scott, D., Lattice Theory, Data Types and Formal Semantics, Formal Semantics of Programming Languages [R. Rustin (Ed.)], Prentice-Hall, Englewood Cliffs, NJ (1972), 65–106.

84. Scott, M. L. Programming Language Pragmatics, 2nd ed. — Morgan Kaufmann publications, 2006. — 856 pp.

85. Sebesta, R. W. Concepts of Programming Languages, 10th ed. — Pearson/Addison-Wesley, 2012. — 814 pp.

86. Sethi, R. Programming Languages: concepts & constructs, 2nd ed. — Addison-Wesley, 1996. — 640 pp.

87. Sommerville, I. Software Engineering. 9th ed. — Addison-Wesley, 2011. — 773 pp.

88. Strachey, C. Fundamental Concepts in Programming Languages, Higher-Order and Symbolic Computation, Vol. 13, Issue 1/2 (April 2000) Special Issue in memory of Christopher Strachey, 11–49.

89. Stroustrup, B., A History of C++:1979–1991, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 271–297.

90. Tanenbaum, A. Modern Operating Systems, 3rd ed. — Prentice Hall, 2008. — 1104 pp.

91. Walls, C. Spring in Action, 3rd ed. — Manning, 2011. — 423 pp.

92. Wirth, N., Recollections About the Development of Pascal, ACM History of Programming Languages Conference II, Cambridge, MA (April1993) [SIGPLAN Notices (28)3 (March 1993)], 333–342.

93. Wirth, N. Systematic Programming: an Introduction. — Prentice-Hall, Englewood Cliffs, NJ, 1973.

94. Wirth, N. Algorithms + Data Structures = Programs. — Prentice-Hall, Englewood Cliffs, NJ, 1976.

95. Wyatt, B, Kavi K., and Hufnagel S., Parallelism in Object-Oriented Languages: A Survey, IEEE Software (9)6 (1992), 56–65.

Алфавитный указатель

Л–О

П–С

Т–Я