Real Time Languages:

Design and Development

Stephen J. Young, M. A., Ph. D.

Computation Department University of Manchester Institute of Science and Technology

Ellis Horwood Limited Publishers Chichester Halsted Press: a division of John Wiley & Sons

New York • Brisbane • Chichester • Toronto 1982

С.Янг

Алгоритмические языки реального времени

Конструирование и разработка

Перевод с английского
Л. В. УХОВА

под редакцией В. В. МАРТЫНЮКА

Москва «Мир» 1986

ББК 32.973
Я 60 УДК 681.3

Янг С.
Я 60 Алгоритмические языки реального времени: конструиро- вание и разработка: Пер. с англ. — М.: Мир, 1985. — 400 с., ил.

Книга написана известным американским специалистом по программированию Она знакомит читателей в доступной форме и з то же время на достаточно высоком научном уровне с основными принципами построения языков программирования реального времени. Подробно разобраны применения этих принципов для языков Модула и Ада. Книга позволяет быстро ориентироваться в языках реального времени и их модификациях.

Для программистов и разработчиков математического обеспечения ЭВМ.

1702070000—215 41-85, ч.1 **ББК**
041(01)-85

6Ф7.3

Редакция литературы по *математическим наукам*

Стивен Дж. Янг
АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ РЕАЛЬНОГО ВРЕМЕНИ!
Конструирование и разработка

Научный редактор Л. Н. Вабынина. Младший научн. редактор Э. И. Качулина Художественный редактор В. И. Шаповалов. Художник Н. М. Иванов. Технический редактор А. Л. Гулина. Корректор Л. В. Байкова

ИБ № 5105

Сдано в набор 04.07.84. Подписано к печати 22.02.85. Формат 60X90'/ie. Бумага книжно-журнальная. Объем бум. л. 12,50.. Гарнитура литературная. Печать высокая. Усл-печ л. 25,00.

Усл. кр.-отт. 25,00. Уч.-изд. л. 23,57. Изд. № 1/3373. Тираж 24 000 экз. Зак. NJ 287. Цена 2 руб.

ИЗДАТЕЛЬСТВО «МИР»
129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29.

С 1982 by Ellis Horwood Limited
С Перевод на русский язык, «Мир», 1985

ОТ РЕДАКТОРА ПЕРЕВОДА

Управление различными объектами в реальном времени является в настоящее время одним из основных применений ЭВМ. В последнее десятилетие появилось очень много управляющих систем различной ориентации, и поэтому весьма актуальными становятся специфические требования к средствам программирования, и в частности требование, чтобы языковые средства адекватно обеспечивали удобство и надежность программирования и последующей эксплуатации таких систем. Практика показала, что для этого нужно существенно больше выразительных возможностей, чем предоставляют такие традиционные языки, как Алгол и Фортран.

На протяжении ряда лет ведется интенсивная теоретическая и прикладная деятельность, нацеленная на разработку и внедрение языков программирования, ориентированных на различные системы управления реальными процессами. Основным теоретическим результатом в этом направлении, по-видимому, следует признать язык программирования Ада, однако к настоящему времени этот язык еще не получил массового практического внедрения. К тому же сам язык Ада все еще находится в стадии становления, и можно ожидать в последующие годы появления в нем дальнейших изменений.

В то же время сложилась определенная методология эффективного конструирования алгоритмических языков реального времени, основывающаяся на ряде принципов, инвариантных относительно конкретного языкового воплощения. Представляется актуальной задача ознакомления с этой методологией и приобщения к этим принципам многочисленных специалистов различной квалификации, вовлеченных в проблематику программирования управляющих систем.

Успешным шагом в этом направлении является предлагаемая книга. В ней в доступной форме излагаются основные принципы конструирования языков программирования управления в реальном времени и разбирается применение этих принципов в трех языках: RTL/2, Модула и Ада, представляющих

последовательные этапы эволюции языков реального времени. Достаточно подробное описание этих языков сопровождается детальными примерами их использования.

Особое внимание уделяется языку Ада. Убедительно демонстрируется, что этот язык позволяет удобно и надежно программировать сложные системы управления реальными процессами. В значительной мере материал книги можно рассматривать как анализ фундаментальных свойств языка Ада.

В книге удачно сочетаются достаточно популярная форма и несомненная глубина изложения. Автор не только описывает конкретные алгоритмические языки, но и дает определенный круг систематизированных общих представлений о необходимых свойствах современных языков реального времени и средствах их достижения. Это полезно для ориентации разработчиков управляющих систем в имеющемся разнообразии доступных языков программирования и их модификаций. Это особенно важно потому, что в настоящее время еще рано говорить о стабильности языков реального времени.

При подготовке русского издания книги возникли проблемы, связанные с терминологией, которую в настоящее время нельзя считать устоявшейся. Мы в основном придерживались терминологии, принятой в переведенных на русский язык книгах: «Язык программирования Ада» (М.: Финансы и статистика, 1981) и П. Вегнер «Программирование на языке Ада» (М.:Мир, 1983). Отметим также, что в примерах программ для большей наглядности идентификаторы и другие выделяемые курсивом слова, как правило, переведены на русский язык.

В. В. Мартынюк

Посвящается Джейн

ПРЕДИСЛОВИЕ

Использование цифровых вычислительных машин характеризуется в последние годы нарастающим расширением области их применений. Но нигде это расширение не было столь впечатляющим, как в системах реального времени. Работающие в режиме реального времени вычислительные машины постоянно используются на телефонных станциях, в промышленных процессах, управлении авиарейсами, в системах спутниковой связи, медицине и т. д. По мере усложнения области применений и повышения стоимости разработки программного обеспечения крайне необходимым становится улучшение средств программирования. В частности, возникает большая необходимость в языках программирования, которые не только обладают достаточно выразительной силой для описания конкретного задания, но и обеспечивают поддержку для построения надежного и легко сопровождаемого программного обеспечения для работы в реальном времени.

Эта книга посвящена проектированию и разработке таких языков. Основной материал излагается в двух частях книги. В первой части подробно обсуждаются аспекты проектирования. В отдельных главах разбираются типизация данных, структурирование данных, структурирование программ, модули, мультипрограммирование, программирование для устройств низкого уровня и обработка ошибок. Везде особое внимание уделяется развитию особо надежных средств, сводящих к минимуму возможность появления программистских ошибок. Во второй части этой книги иллюстрируется разработка практических языков реального времени путем подробного описания трех конкретных языков: RTL/-2, Модула и Ада. В каждом случае описываются свойства языка, а затем приводится развернутый пример его использования. Далее следует обсуждение языка с точки зрения критериев, изложенных в первой части книги.

Главная цель этой книги — дать ясное представление о факторах, определяющих проектирование современного языка

реального времени. В этом отношении она будет интересна студентам, изучающим проектирование языков программирования, а также студентам и программистам с прикладной ориентацией, которые изучают и применяют на практике современные языки реального времени, и в частности тем, кто изучает и использует язык Ада.

И наконец, я хочу выразить благодарность всем, кто помог мне конструктивными замечаниями во время написания этой книги, и особенно Бэтти Шарплс и моей жене Джейн, напечатавшим рукопись.

18 ноября 1981 *С. Дж. Янг*

Часть I. ПРОЕКТИРОВАНИЕ

Глава 1.
ТРЕБОВАНИЯ К ПРОЕКТИРОВАНИЮ ЯЗЫКОВ РЕАЛЬНОГО ВРЕМЕНИ

1.1. ХАРАКТЕРИСТИКИ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Цель первой части книги — обсудить проектирование языков программирования, удобных для использования в областях применения систем реального времени. В данной вводной главе будут выявлены характеристики систем реального времени, на основе которых можно будет установить некоторые базовые критерии проектирования и составить список требований к языку.

Прежде чем приступить к обсуждению характеристик систем реального времени, стоит определить, что понимается под термином «реальное время», принимая во внимание, в частности, тот факт, что этот термин часто используется в самых разнообразных контекстах с соответственно различными значениями. В самом широком смысле термин «реальное время» можно использовать применительно к деятельности или системе по обработке информации, когда требуется отвечать на поступающие извне входные сигналы, причем задержка ответа должна быть конечной и не превышающей специфицированного значения. Таким образом, например, вычислительные средства общего назначения, с которыми пользователи общаются через интерактивные терминалы в режиме on-line, являются системой реального времени. Когда пользователь вводит команду, он надеется, что получит ответ не позднее, чем через несколько секунд, и хорошо спроектированная система должна оправдывать эту надежду. Тем не менее если ответ не пришел даже, скажем, через две минуты, то обычно систему не считают непригодной. Пользователь вправе быть недовольным, но если в конечном счете ожидаемый результат получен, то система функционирует правильно.

Впрочем, это не всегда так. Для некоторого класса работ в реальном времени вычисление результата за время, большее требуемого, вполне может быть приравнено к вычислению неверного результата. В этом классе применения вычислительная машина связана непосредственно с некоторым физическим устройством или комплексом и должна обеспечивать управление

или контроль над производственным процессом этого комплекса. Типичными примерами подобного применения являются лабораторные измерения, телекоммуникационное обслуживание, управление процессами, управление полетом ракеты, аэронавигация и управление промышленными роботами. Ключевым моментом во всех этих применениях является то, что главная задача *вычислительной* машины состоит в выполнении функций информа-

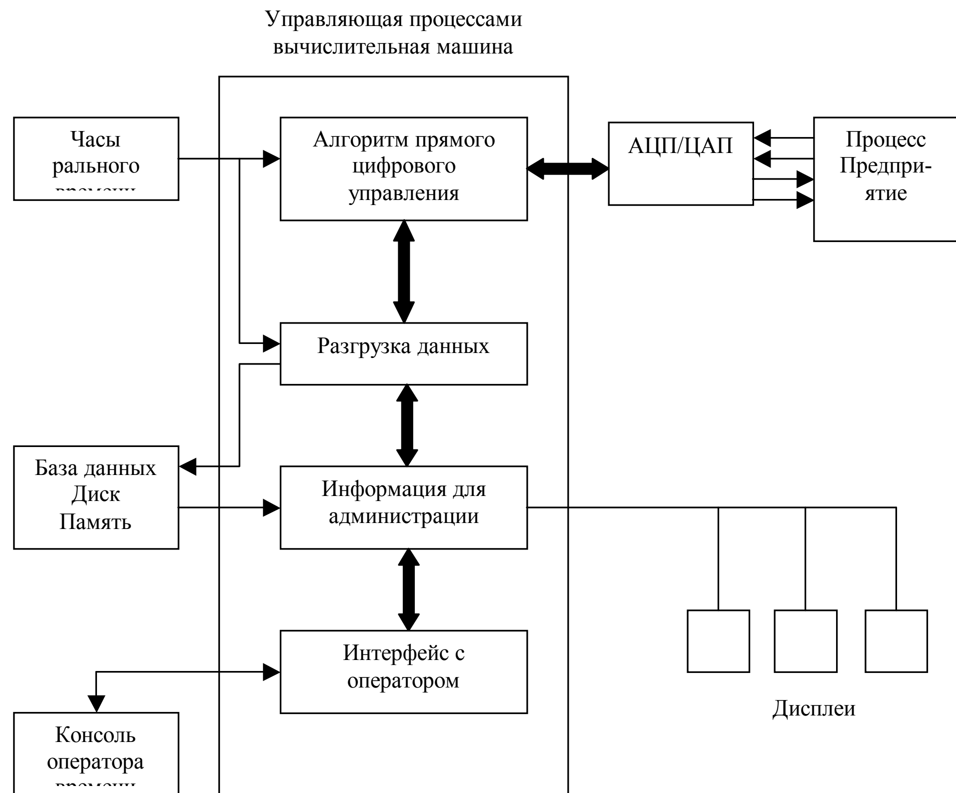


Рис. 1.1. Типичная небольшая система управления процессами.

ционной компоненты в рамках объемлющей инженерной системы. По этой причине такие области применения стали называться *системами реального времени со встроенной вычислительной машиной* (embedded computer real time systems), поскольку вычислительная машина встроена в большую систему. Именно для этого класса встроенных систем и разрабатывались в первую очередь языки программирования в реальном времени, и поэтому термин «реальное время» мы будем использовать в связи с этим видом прикладной деятельности.

Чтобы лучше понять природу таких систем, рассмотрим типичную небольшую систему управления процессами, изображенную схематически на рис. 1.1. В этой системе на вычислительную машину возлагается задача управления некоторыми

промышленными процессами. Система непосредственно связана с производством с помощью аналого-цифровых преобразователей (АЦП), которые позволяют ей измерять характеристики состояния производства и осуществлять управление промышленными операциями. Так как используемые алгоритмы управления требуют периодической выборки данных о состоянии производства, система оснащается часами (измеряющими реальное время), которые прерывают работу вычислительной машины в момент выборки данных. Наряду с необходимостью интерфейса с производством вычислительной машине необходим также интерфейс с операторами, которые инициируют операции начала и завершения и могут варьировать общее протекание производственного процесса. Кроме этого, инженеры - наладчики и администрация также должны иметь возможность оказывать влияние на производственную деятельность. Для обеспечения этого требования выполняется операция запоминания информации о производственной деятельности в базе данных на диске. Доступ к этой информации осуществляется с помощью ряда дисплеев, которые в современных системах являются главным образом устройствами цветного графического вывода.

Программное обеспечение такой системы естественно разделяется на программы, выполняющие отдельные функции, или задачи. В нашем конкретном примере выделяются четыре задачи: для выполнения прямого цифрового управления (ПЦУ), для запоминания данных, для обеспечения администрации информацией и для обеспечения интерфейса с оператором соответственно. В других системах задач может быть больше. Так, например, могут потребоваться программы для аварийного управления, долгосрочной оптимизации, супервизорного управления и т. д. Кроме этого, может потребоваться создавать временные задачи, позволяющие развивать программное обеспечение на фоне работающего предприятия.

Каждая из этих задач представляется в программном обеспечении в виде процесса. Процессы выполняются параллельно и взаимодействуют при необходимости, синхронизируясь по определенным событиям и обмениваясь информацией. Разбиение программного обеспечения на процессы упрощает проектирование и разработку и облегчает удовлетворение требований ко времени ответа системы. Например, при получении прерывания от часов реального времени задача ПЦУ должна обеспечить немедленный ответ, измерив характеристики нового состояния производства и вычислив соответствующие выходные данные управления. Выполнению этой задачи высокого приоритета не должно мешать выполнение в системе других задач более низкого приоритета. Если программное обеспечение в целом

спроектировано как одна последовательная программа, то реализация приоритетов для различных функций обработки окажется гораздо более трудным делом.

Рассмотрев общую схему типичной системы, мы можем перечислить характерные особенности программной системы реального времени. Во-первых, она должна быть исключительно надежной. Сбой такой системы может иметь весьма тяжелые последствия либо в отношении производства продукции в случае управления процессом, либо в отношении человеческих жизней в таких особых случаях, как аэронавигация или управление работой атомной станции. Во-вторых, такая система обычно велика и сложна. Вместе с проблемой надежности это означает, что стоимость ее разработки и поддержания может оказаться очень высокой. В-третьих, она должна реагировать на многочисленные и разнообразные внешние события с гарантированным временем ответа. В-четвертых, в ней должен быть обеспечен интерфейс, как с широким спектром нестандартных устройств ввода-вывода, так и с более стандартными периферийными устройствами вычислительной машины. В-пятых, она должна быть эффективной, так чтобы имеющееся аппаратное обеспечение использовалось наилучшим образом.

Ясно, что язык реального времени, обеспечивающий эффективную поддержку проектирования, разработки и сопровождения такого программного обеспечения, должен удовлетворять многим требованиям. В следующем разделе обсуждаются общие критерии проектирования, и затем в разд. 1.3 на основе этих критериев и практических характеристик систем реального времени выводится ряд существенных требований к языку.

И наконец, следует подчеркнуть, что, хотя язык реального времени разрабатывается главным образом для удовлетворения нужд программирования систем реального времени, его использование редко ограничивается только этой областью. Современный язык реального времени наряду со специфическими возможностями должен обладать всеми возможностями языка общего назначения. Поэтому эти языки идеальны для применения в таких областях системного программного обеспечения, как операционные системы, компиляторы, связывающие программы, препроцессоры и т. д. В спецификациях системы реального времени часто содержится требование, чтобы все вспомогательное программное обеспечение (включая компилятор) писалось на том же языке, что и прикладное программное обеспечение. Это уменьшает общие расходы и сводит обучение программиста к изучению лишь одного языка. Большая часть материала этой книги применима также и к проектированию языков программирования общего назначения.

1.2. ОБЩИЕ КРИТЕРИИ ПРОЕКТИРОВАНИЯ

В данном разделе в качестве основы проектирования языков реального времени предлагается шесть основных критериев. Эти критерии носят названия: надежность, удобочитаемость, гибкость, простота, мобильность и эффективность.

1.2.1. Надежность

Надежность при проектировании языка является мерой степени автоматического обнаружения ошибок, которое может быть выполнено либо компилятором, либо системой, поддерживающей выполнение скомпилированной программы. Как было указано в предыдущем разделе, программное обеспечение реального времени должно функционировать надежно даже с учетом невозможности исчерпывающего тестирования. Отсюда ясно, что внутренняя надежность языка играет жизненно важную роль при создании надежных программ.

В общем случае проект языка должен быть таким, чтобы как можно больше ошибок обнаруживалось во время компилирования, а не во время работы скомпилированной программы. Это желательно по двум причинам. Во-первых, чем раньше при разработке процесса обнаружена ошибка, тем меньше общая стоимость. Компиляция может быть выполнена на любой воспринимающей язык машине, тогда как тестирование при прогоне программы должно выполняться на фактическом целевом машинном оборудовании (или при его моделировании). Следовательно, тестирование прогоном часто будет очень дорогим, особенно когда оно приводит к прерыванию работ отдельных средств оборудования или всего производства. Во-вторых, для обнаружения ошибок во время прогона часто требуется внесение компилятором в выходной объектный код добавочных контрольных команд. Это естественно приводит к дополнительным расходам, как в терминах скорости выполнения, так и в терминах использования памяти, что в обоих случаях не способствует эффективному функционированию в реальном времени.

Принципиальным средством достижения высокой надежности компиляционного уровня является система типизации данных; подробно она обсуждается в гл. 2 и 3, здесь мы рассмотрим один простой пример. Предположим, что массив a, состоящий из 20 целых чисел, индексируется целой переменной i таким образом, что i-*я* компонента массива a обознача- ется a [i]. Очевидно, что надежный язык должен обеспечить выполнение условия $1 \le i \le 20$ в любом месте программы, где встречается $a[i]$. Для этого имеются две возможности. Тради-

ционный подход заключается в простом внесении в готовую программу перед каждым обращением к массиву дополнительного кода, проверяющего выполнение наших условий. Более современный подход состоит в проверке значения i не во время его использования, а скорее во время присваивания значения переменной i. Это требует добавочных усилий программиста, который должен специфицировать область значений при описании переменной. Зная, что тип переменной i ограничивает возможные значения областью от 1 до 20, нам нет нужды вообще проверять значения i при обращении к $a[i]$. Вместо этого проверяются присваиваемые i значения. Как ни удивительно, очень часто законность таких присваиваний может быть проверена во время компиляции, так что общий объем фактически необходимого объектного кода проверки существенно уменьшается. Еще одно существенное преимущество этого подхода состоит в том, что при этом значительно увеличивается ясность программ, так как область значений, принимаемых каждой переменной, устанавливается явно.

И, наконец, следует подчеркнуть, что безусловно имеется предел числа ошибок, которые могут быть обнаружены любой языковой системой. Например, ошибки в логическом построении программы автоматически не могут быть обнаружены. Впрочем, ошибки такого рода будут случаться реже, если сам язык поощряет программиста писать ясные, хорошо структурированные алгоритмы, предоставляя ему возможность выбора подходящих языковых конструкций. Отсюда следует, что надежный язык должен также быть хорошо структурированным и удобочитаемым языком.

1.2.2. Удобочитаемость

Удобочитаемость языка зависит от широкого спектра факторов, включающего с одной стороны выбор ключевых слов, а с другой — возможности модуляризации программ. Главная задача состоит в выборе такой четкой нотации языка, которая позволяла бы при чтении текста программы легко выделять основные понятия каждой конкретной части программы, не обращаясь к сопровождающим программу блок-схемам и словесным описаниям.

Высокая степень удобочитаемости оказывается полезной с различных точек зрения. Во-первых, уменьшается стоимость документирования, если центральным элементом документации является сама программа. Это весьма справедливо, в частности, для проектов программного обеспечения с большим временем существования, когда поддержание обновляемой сопроводительной документации в условиях неизбежного множества

последовательных модификаций может оказаться весьма трудным делом. Во-вторых, как замечено выше, хорошая удобочитаемость позволяет легче понимать программу и, следовательно, быстрее находить ошибки. В результате увеличивается надежность. В-третьих, хорошая удобочитаемость позволяет легче сопровождать программу. Когда требуется внести изменения в программу, часто бывает так, что сотрудник, которому поручено это сделать, не принимал участия в ее первоначальной разработке. Ясно, что существенные изменения могут быть сделаны лишь тогда, когда работа программы понимается совершенно правильно.

Очевидно, что реализация требований хорошей удобочитаемости зависит от самого программиста, который должен постараться по возможности четче структурировать свою программу и так расположить ее текст, чтобы подчеркнуть эту структуру. Тем не менее, важную роль играет и структура используемого программистом языка. На нижних уровнях программных конструкций язык должен обеспечить возможности четкой спецификации того, какие объекты данных подвергаются обработке и как они используются. Эта информация определяется выбором идентификаторов и спецификаций типов данных. Программиста не нужно заставлять прибегать к искусственным построениям, вводя в язык такие ограничения, как максимальная длина идентификатора или определенные фиксированные типы данных. Алгоритмические структуры должны выражаться в терминах легко понимаемых "структур управления, таких, как **if ... then ..., else** ... и т. д. Ключевые слова не следует сводить к аббревиатурам, а символы операций нужно выбирать аккуратно, так, чтобы они отображали их смысл. Случайные читатели, знающие язык лишь поверхностно, все же должны понимать существо программы, не обращаясь к помощи руководства по используемому языку. На более высоких уровнях программных конструкций язык должен обеспечивать возможности модуляризации и управления областями действия имен. Общее поведение программы гораздо легче понять, когда она составлена из ряда автономных операционных единиц, каждая из которых должна быть понята вне связи с остальными частями программы. Это имеет особое значение тогда, когда программе предстоит пройти этап модификации. Если текст программы прямо указывает на ее логическую структуру, то влияние любого изменения на всю программу гораздо легче проследить, так как нужно исследовать подробно только ту логическую единицу, в которую вносится изменение.

И наконец, следует принять как должное увеличение длины программы, являющееся неизбежной платой за выполнение требования хорошей удобочитаемости. Впрочем, некая много-

словность — это малая плата за все остальные выгоды, особенно если вспомнить, что программа пишется лишь один раз, а читается в общем-то многократно.

1.2.3. Гибкость

Язык должен предоставить программисту достаточно различных возможностей для выражения всех операций, которые требуются в программе, не заставляя его прибегать к вставкам машинного кода или различным ухищрениям. Критерий гибкости особенно существен в режиме реального времени, где, возможно, потребуется работать с широким спектром нестандартного периферийного оборудования. Для достижения высокой гибкости приходится идти на компромисс с требованиями надежности. Как правило, при разработке языка обеспечивается достаточная гибкость, соответствующая выбранной области применения и не больше.

1.2.4. Простота

Требование простоты в проекте языка не нуждается в особом обосновании. Простота уменьшает затраты на обучение программиста и уменьшает вероятность совершения программистских ошибок, возникающих в результате неправильной интерпретации спецификаций языка. Обычно простота языка уменьшает и размер компилятора, облегчает получение более эффективного объектного кода и увеличивает мобильность компилятора.

Для достижения простоты не обязательно избегать сложных языковых механизмов, скорее при введении некоторой языковой особенности нужно стараться не накладывать случайные ограничения на ее использование. Так, например, при реализации механизма массивов следует дать программисту возможность объявлять массив любого типа данных, допускаемого языком. Если же, наоборот, в языке запрещается использование массивов некоторого типа, то в результате язык окажется скорее более сложным, чем простым. В общем случае основные правила языка изучить легко, трудно запомнить связанные с ними списки условий и ограничений на их применение.

1.2.5. Мобильность

Как и в случае критерия простоты, необходимость мобильности также очевидна. Проектирование языка, независимое от базового машинного оборудования, дает возможность переносить программное обеспечение с машины на машину с относи-

тельной легкостью. Это позволяет высокую стоимость программного обеспечения распределить на целый ряд машинных конфигураций.

На практике мобильности достичь довольно трудно, особенно в системах реального времени, где одна из противоречивых задач состоит в извлечении максимальных выгод из базового машинного оборудования. В этом отношении особенно трудно поддаются решению проблемы, связанные с различающимися длинами слов памяти.

1.2.6. Эффективность

Системы реального времени часто должны обеспечивать высокую вычислительную пропускную способность, чтобы не нарушить ограничений, налагаемых управляемым ими внешним оборудованием. Поэтому язык должен быть построен так, чтобы его можно было реализовать эффективно. Более того, поскольку необходимо гарантировать определенное время ответа, следует избегать языковых конструкций, ведущих к непредсказуемым издержкам прогона программ (например, сборки мусора в схеме динамического распределения памяти).

При разработке первых языков реального времени необходимость эффективности рассматривалась как фактор первоочередной важности, в результате чего приходилось жертвовать факторами надежности, гибкости и простоты. Однако теперь все снижающаяся стоимость машинного оборудования и все возрастающая стоимость программного обеспечения позволяют считать, что необходимость эффективности больше не является критичной. Поэтому основной целью при разработке современного языка является обеспечить возможность написания надежных и недорогих программ. Хотя, безусловно, необходимость эффективной реализации сохраняется, она постепенно отходит на второй план.

1.3. ТРЕБОВАНИЯ К ЯЗЫКУ ДЛЯ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Описанные в предыдущем разделе шесть основных критериев проектирования языка реального времени перечислены в приблизительном порядке их важности. Отсюда следует, что требования высокой надежности и хорошей удобочитаемости считаются совершенно необходимыми для успешного применения языка в программном обеспечении реального времени.

Хотя необходимость высокой надежности оказывает большое влияние на все аспекты проектирования языка, основой надежного языка является система его типов данных. Тип данных

специфицирует область значений, которые может принимать объект этого типа, и множество операций, применимых к объектам этого типа. Поэтому тип данных определяет, что представляет собой объект, и накладывает ограничения на то, как он может использоваться. Язык реального времени должен иметь логичную систему типов данных, так чтобы появляющиеся во время манипуляции данными ошибки можно было обнаружить, а также чтобы можно было однозначно специфицировать предполагаемый вид использования данных. Типизация данных подробно обсуждается в гл. 2 и 3.

Наряду с необходимостью спецификации данных необходимо также точно специфицировать действия, которые должна выполнять программа. В последние годы выгоды «структурного программирования» признаются всеми. Чтобы язык помогал написанию хорошо структурированных программ, он должен содержать конструкции для представления различных условных и итеративных структур управления и для объединения логически соотносящихся действий в процедуры и функции. Эти классические программные структуры рассматриваются в гл. 4. Впрочем, как было отмечено ранее, одной из характеристик систем реального времени является то, что они часто бывают большими и сложными. Чтобы удовлетворить требованиям «крупномасштабного программирования», язык реального времени должен также предоставить механизм для модуляризации. Этот аспект структурирования программ изложен в гл. 5.

В свете того факта, что реальные системы, естественно, описываются как множество одновременно выполняемых задач, язык реального времени должен предоставлять некоторые возможности для мультипрограммирования. Это может быть сделано либо путем спецификации стандартного интерфейса с мультипрограммной операционной системой, либо путем предоставления в самом языке возможностей описания мультипроцессорных систем. Данный аспект проектирования языка реаль- ного времени обсуждается в гл. 6.

Одной из уникальных характеристик программного обеспечения реального времени является необходимость работы с нестандартными устройствами ввода-вывода. Отсюда следует, что язык должен обеспечить некоторые возможности прямого программирования работы устройств низкого уровня. Традиционно это осуществляется путем разрешения непосредственного внесения машинного кода в код более высокого уровня. Впрочем, в гл. 7 описывается метод программирования устройства высокого уровня, который не требует обращения к машинному коду.

И наконец, для достижения как можно более высокой надежности программное обеспечение реального времени должно

обладать способностью справляться с ситуациями, когда проявляются ошибки. Языковые возможности для обработки ошибок описываются в гл. 8.

Таким образом, в следующих семи главах обсуждаются основные требования к языкам программирования в реальном времени. В заключительной главе первой части книги представлено резюме основных факторов, проанализированных в каждой из этих глав, и обсуждается их интеграция в рамках одного языка. Где это возможно, лежащие в основе принятых решений принципы прослеживаются достаточно глубоко. Задача состоит не просто в выборе хорошего проекта языка, но скорее в объяснении того, как мы приходим к этому проекту и на какие при этом нам приходится идти компромиссы. При таком подходе закладываются основы для гораздо более глубокого понимания проектирования языков программирования. Это понимание позволит нам более аккуратно оценить достоинства существующих языков и легче изучить новые языки.

Глава 2.
ПРОСТЫЕ ТИПЫ ДАННЫХ

2.1. ОСНОВНЫЕ ПОНЯТИЯ

Фактически все современные цифровые вычислительные машины базируются на архитектуре фон Неймана, где двумя главными компонентами являются центральное процессорное устройство (ЦПУ) и память. Память состоит из массива адресуемых элементов двоичных данных, называемых словами. Работа вычислительной машины заключается в выборе слов данных из памяти в ЦПУ, модификации их содержимого и записи слов обратно в память. По существу, вычисление можно рассматривать как отображение некоторого начального состояния в некоторое требуемое конечное состояние, причем данные в памяти изменяются последовательно по одному слову за раз.

Вид информации, представленный каждым словом данных в памяти, может быть различен. Например, некоторые слова могут содержать целые числа, в то время как другие могут содержать символы. Программирование на машинном уровне требует точного знания того, как эти данные представлены в виде последовательности битов и какие машинные команды должны применяться для реализации требуемых операций. Это трудоемкий и чреватый ошибками процесс, обойтись без которого можно путем применения языка высокого уровня.

Язык высокого уровня дает программисту абстрактную модель, в которой данные и операции над данными можно специ-

фицировать в проблемно-ориентированных, а не в машинно-ориентированных терминах. Это упрощает процесс разработки программы и, соответственно, уменьшает количество совершаемых ошибок. Более конкретно, язык высокого уровня обеспечивает следующие возможности доступа к объектам данных в памяти и их модификации:

(i) На объекты данных ссылаются с помощью определенных пользователем имен, а не конкретных адресов памяти.

(ii) Объекты данных связаны с типом, определяющим множество значений, которые могут приниматься объектами этого типа, и множество операций, которые могут применяться к объектам этого типа.

В ранних языках введение понятия типа данных было чисто прагматическим приемом, к которому прибегали с целью облегчить работу компилятору. И на самом деле пользователь обращал мало внимания на понятие типа. Для современных языков разработаны очень сложные механизмы типов, позволяющие программисту выразить решение задачи с большей ясностью и со значительно более высокой надежностью. Хороший механизм типов на самом деле является ключевым фактором при обеспечении надежности языка программирования, а, как указывалось ранее, необходимость надежности имеет первостепенную важность при программировании в реальном времени.

Чтобы понять, почему типизация данных так важна для обеспечения надежности, рассмотрим некоторые классы ошибок, встречающиеся в типичных программах:

(i) При обращении к объектам данных могут использоваться неверные имена.

(ii) Объектам данных могут присваиваться логически некорректные значения. Так, например, если объект данных используется для подсчета числа появления некоторого события, то присваивание этому объекту отрицательного значения будет некорректным.

(iii) К объекту данных может быть применена логически некорректная операция. Так, например, если два объекта данных используются для хранения символьных значений, то выполнение над ними операции сложения будет некорректным.

Ошибки каждого из этих классов могут возникнуть в результате ошибок человека при печати, небрежности или логических ошибок, допущенных при разработке программы.

Если учесть трудности тестирования программ, работающих в реальном времени, то становится совершенно очевидным, почему проект языка обязательно должен предусматривать возможность обнаружения большинства этих ошибок либо

компилятором, либо во время работы программы. Впрочем, проверка программы при прогоне уменьшает эффективность программы реального времени, и поэтому, где это возможно, нужно предпочесть проверку во время компиляции. Способом,, обеспечивающим такую проверку во время компиляции, является ограничение множества значений и операций, допускаемых для каждого объекта данных, до минимального требуемого. Как будет показано в следующем разделе, это может быть сделано при оснащении языка подходящей системой типов данных.

Безусловно, высокая надежность не единственное требование к языку программирования реального времени. Язык должен также обеспечить адекватное множество типов данных и операций для выражения решения задач в выбранной области. В общем случае в языке имеется множество предопределенных типов данных и операций и один или более механизмов для спецификации типов, определяемых пользователем. Следующие четыре предопределенных типа данных являются общими для большинства языков:

(i) целый
(ii) плавающий
(iii) логический
(iv) символьный

Кроме того, язык реального времени должен иметь еще один числовой тип

(v) фиксированный

для выполнения с высокой скоростью операций вещественной арифметики на машинах, для которых аппаратно не реализованы операции с фиксированной запятой.

Все эти предопределенные типы являются простыми типами. У объектов простого типа нет внутренней структуры, и они могут содержать лишь одно значение. Но языки могут также позволять определять структурные типы. Объекты структурного типа состоят из других простых и (или) структурных типов и могут содержать составные значения.

В этой главе обсуждаются простые типы, рассмотрение структурных типов откладывается до гл. 3. После обсуждения в следующем разделе механизмов типизации в разд. 2.3 описываются перечислимые типы, которые включают предопределенные логические и символьные типы, а в разд. 2.4 описываются числовые типы. И, наконец, в разд. 2.5 кратко говорится о разработке типов данных в реальных языках.

2.2. МЕХАНИЗМЫ ТИПИЗАЦИИ

В данном разделе исследуются средства, с помощью которых понятие типа данных может быть использовано для увеличения ясности и надежности программы. Типизация в разработке языка представляет собой сложный процесс, является объектом многих ведущихся в настоящее время исследований и вполне заслуживает самостоятельной книги. Чтобы не перегрузить читателя описанием большого числа различных подходов к типизации данных, мы рассмотрим подробно лишь один. Этот подход хорошо иллюстрирует современное понимание вопроса типизации в разработке языков реального времени.

Как и во всей книге, используемые обозначения соответствуют стилю синтаксиса языка Паскаль. Не знакомые с языком Паскаль читатели могут обратится за справками к одному из стандартных учебников (например [1], Йенсен и Вирт, 1979; Уэлш и Элдер, 1979). Впрочем, мы надеемся, что обозначения достаточно понятны и сами по себе.

2.2.1. Слабая типизация

Основа надежности языка заключается в возможности проверить, что каждая операция, выполняемая над объектом данных, соответствует специфицированному типу этого объекта. Проекты ранних языков были слабо типизированными. Это означает, что информация о типе использовалась только для обеспечения корректности на машинном уровне. Существуют три особенности этой стратегии типизации.

Во-первых, операция, которая может восприниматься машиной как корректная, может быть некорректной на абстрактном уровне программы. Например, предположим, что символьная переменная объявлена в описании

var *c*: *сим*;

тогда слабо типизированный язык вполне может допустить запись

c:= 4

так как на машинном уровне символы представлены целыми числами в диапазоне от 0 до 255. Следовательно, для машины это корректно. Однако на абстрактном уровне программы это не так. Возможно, именно это программист имел в виду, но

[1] На русский язык переведена также книга П. Грогоно «Программирование на языке Паскаль»,— М.: Мир, 1982.—*Прим. ред.*

вполне вероятно, он допустил ошибку, намереваясь, например, написать

c:= '4'

Во-вторых, при работе с разными типами слабо типизированный язык для сохранения корректности предусматривает выполнение операции преобразования типа. Например, если встретится

var *x, y* : *вещественный*;
i, j, k : *целый*;

то присваивание *i* := *x* потребует преобразования вещественного объекта *x* перед его присваиванием объекту *i* из его внутреннего представления с плавающей запятой в целое представление. Разумеется, эта операция приведет к ошибке, если значение *x* выходит из реализуемой области целых чисел, но предупреждения об этом в тексте программы не имеется. Еще одна проблема, связанная с неявным преобразованием типа, состоит в необходимости совершенного знания языка для точного определения порядка преобразований типов, выполняемых в произвольно составленных выражениях. Так, например, непонятно, какую операцию преобразования нужно выполнять сначала в операторе присваивания

k := *x* - *j*

Нужно ли сразу *x* преобразовывать в целый тип, или сначала *j* преобразовать в вещественный тип, а затем *x* — *j* преобразовать обратно в целый? Заметим, что если *x* превосходит максимальное целое значение, то первое безусловно приведет к ошибке, а во втором случае этого может и не произойти. В руководстве по языку можно найти правила поведения в этой ситуации, но тем не менее здесь можно легко ошибиться или неверно интерпретировать написанную другим программу.

В-третьих, в системе со слабой типизацией очень соблазнительно разрешить для некоторых типов определение некорректных операций. Это верно, в частности, для языков реального времени, где требуется возможность работы с битами. Например, наряду с нормальными арифметическими операциями можно для целых типов определить операции логического сложения и операции сдвига. Предположим, что определены операция логического «или» **or** и операция сдвига **shl**, такая, что *i* shl *n* означает сдвиг целого *i* на *n* позиций влево. Тогда в предположении, что длина слова равна 16 битам, оператор присваивания

i := (*k* **shl** 12) **or** 1

можно использовать для упаковки значения *k в* четырех левых разрядах *i* и засылки l в крайний правый разряд. Эта разновидность операций часто выполняется во время работы с такими устройствами машинного оборудования, как регистры ввода-вывода и регистры управления. Хотя этот вид «расширения типа» увеличивает выразительную мощь языка, он ведет к написанию неясных программ и построению менее строгого и, следовательно, менее надежного компилятора, вынужденного разрешать выполнение расширенных операций. Рассмотренные проблемы характерны для слабо типизированных языков. Программы реального времени должны быть надежны, но их трудно подвергнуть исчерпывающему тестированию. Поэтому увеличение гибкости, обеспечиваемое слабой типизацией, является слишком дорогой ценой за резкое уменьшение ясности программ и необходимость дополнительного контроля во время работы компилятора. Опыт, связанный с появившимся в 1971 году языком программирования Паскаль и большим количеством разработанных позднее проектов языков, показал, что «сильная типизация» является лучшей основой проектирования языков. К обсуждению ее мы сейчас и переходим.

2.2.2. Сильная типизация

Сильно типизированный язык это язык, в котором
(i) Каждый объект обладает уникальным типом.
(ii) Каждый тип определяет множество значений и множество операций.
(iii) В каждой операции присваивания тип присваиваемого значения и тип объекта данных, которому производится присваивание, должны быть эквивалентны.
(iv) Каждая применяемая к объекту данных операция должна принадлежать множеству операций, определяемому типом объекта.

Тогда для следующих описаний данных

var *x*: *вещественный*;
i: *целый*;
b: *логический*;
c: *сим*;

в системе с сильной типизацией все перечисленные ниже операторы присваивания незаконны:
(a) *i*:=*'A'* — различные типы левой и правой части
(b) *x*:= *i* — то же самое
(c) *c* := 10 — то же самое
(d) *i* :=*i* **or** *7* — операция **or** принадлежит логическому типу

Ясно, что какие-то присваивания вида (b) необходимы. В сильно типизированном языке соотносящиеся типы должны быть переведены в совместимые применением функции явного преобразования типа, например,

x := *вещественный* (*i*)

где имя типа *вещественный* применяется к целому значению *i* и дает вещественный результат.

Сильная типизация резко повышает надежность и ясность программы. Всем операциям на абстрактном уровне языка обеспечивается корректность. Считается незаконным присваивание значения одного типа данных объекту данных другого типа, даже если внутреннее представление обоих типов идентично. В каждом из таких случаев присваивания должна явно применяться функция преобразования типа. Даже тогда, когда результат ее выполнения эквивалентен тождественному преобразованию, как, например, в операторе

c:=сим(10)

включение функции преобразования необходимо. Оно заставляет программиста недвусмысленно заявить, что он намеренно выполняет необычную операцию и что это не случайная ошибка.

2.2.3. Производные типы

Если все простые объекты данных в программе описаны с помощью предопределенных типов, то от этого существенного увеличения надежности не произойдет. Эффект сильной типизации невелик вследствие малого числа и широкого диапазона значений предопределенных типов. Следующим шагом в разработке хорошего механизма типизации является введение понятия типов данных, определенных пользователем.

На абстрактном уровне языка программирования тип данных можно рассматривать как факторизацию определенных свойств, являющихся общими для конкретного класса объектов. Если множество типов данных ограничено предопределенными типами, то все объекты, которые представлены, скажем, вещественным типом, по необходимости принадлежат одному и тому же классу просто потому, что они одного типа. Это, однако, может не соответствовать смыслу и использованию таких объектов в программе.

Рассмотрим, например, простую программу последовательного чтения значений возраста и вычисления суммарного возраста для группы из десяти человек.

```
program возраст;
        var этот_возраст, абщий_воэраст : целый;
         i : целый;
```

```
begin
   общий_возраст := 0;
   for i:= 1 to 10 do  begin
      читать(этот_возраст);
      общий_возраст := общий_возраст + этот_возраст
   end;
   писать(общий_возраст)
end
```

В данном примере все три переменные *этот_возраст, общий_возраст* и *i* являются целыми, но ясно, что они принадлежат к двум логически разным классам объектов. Если бы программист написал

общий_возраст := *общий_возраст* + *i*

внутри цикла, то это означало бы, что по всей вероятности он допустил ошибку, но компилятор обнаружить ее не смог, так как типы переменных идентичны.

Преимущество сильной типизации заключается в том, что программисту разрешается определять при описании типа свои собственные типы. Программа «возраст» может быть переписана так:

```
program возраст2;
type   возраст = целый;
       индекс = целый;
var  этот, общий : возраст;
          i' индекс;
begin
   общий := 0;
   for i := 1 to 10 do
   begin
     читать(этот);
     общий := общий + этот
   end;
   писать(общий)
end
```

В данном варианте программы определяются два новых типа *возраст* и *индекс* в терминах существующего типа *целый.* Описанные с помощью типа *возраст* объекты помещаются явно в класс, логически отличный от класса объектов, описанных типом *индекс.* Такие новые типы называются производными типами.

Введение в язык производных типов увеличивает его надежность, позволяя компилятору, по крайней мере в принципе, обнаруживать такие ошибки, как

общий := *общий* + *i*

Обнаруживается ли такая ошибка на самом деле, зависит от используемого в языке метода определения эквивалентности типов. В связи с производными типами возникает еще вопрос о том, какие атрибуты наследует производный тип от порождающего типа. Этот вопрос обсуждается в следующих двух разделах.

Заметим, наконец, что использование производных типов увеличивает наряду с надежностью и ясность программ. Имя типа можно выбрать так, чтобы оно отражало вид использования объектов этого типа. Обратите внимание на то, что в приведенной выше программе *возраст2* имена идентификаторов *общий_возраст* и *этот_возраст* записаны в упрощенной форме, так как информация о том, что они содержат «значения возраста», задается теперь в описании производного типа.

2.2.4. Эквивалентность типов

Когда компилятору для сильно типизируемого языка предстоит обрабатывать оператор присваивания вида

x := *выражение*

он разрешит выполнение операции присваивания только в случае эквивалентности типа левой части оператора типу результата вычисленной правой части. Следовательно, у него должно иметься правило вычисления эквивалентности типов. По существу, возможны две различные основы для такого правила: структурная эквивалентность и именная эквивалентность.

При использовании структурной эквивалентности два объекта принадлежат эквивалентным типам, если у них одинаковая структура. Поэтому при использовании структурной эквивалентности компилятор должен разрешать присваивание в таком операторе, как *общий* := *i,* так как структурно как общий, так и *i* целые. Фактически при структурной эквивалентности производные типы не более чем синонимы для имени порождающего типа. При использовании же именной эквивалентности два объекта принадлежат эквивалентным типам, только если они описаны с помощью одного и того же типа. Поэтому при использовании именной эквивалентности компилятор не разрешит выполнение оператора *общий* := *i*. Именная эквивалентность является гораздо более ограничивающей формой сильной типизации, чем структурная эквивалентность. Она

обеспечивает большую надежность компилятора и не нуждается в алгоритмах сравнения шаблонов, которые требуются при реализации структурной эквивалентности для сложных структурных типов данных. Ее недостатком является необходимость явного определения функций преобразования типа и ограничение использования описаний анонимного типа. Последнее объясняется подробно позднее. Структурная эквивалентность, с другой стороны, позволяет более гибко пользоваться аппаратом сильной типизации и легче воспринимается программистами, привыкшими к «слабой типизации». Для производных типов она обеспечивает только большую ясность программы, не увеличивая ее надежности. Зато функции преобразования не требуются для типов с одинаковым порождающим типом, и без ограничений можно пользоваться анонимными типами.

В общем, именная эквивалентность ведет к большей надежности, и в дальнейшем в этой части книги мы будем пользоваться только ею.

И наконец, стоит указать, что приведенные выше рассуждения относятся главным образом к эквивалентности простых типов, являющихся производными от предопределенных типов. Фактически настоящие трудности со структурной эквивалентностью возникают при структурных типах, описываемых в следующей главе.

2.2.5. Наследование атрибутов

Множество значений, литеральные обозначения и определенные для типа операции составляют множество атрибутов типа. Если используется структурная эквивалентность типов, то вопрос о том, какие атрибуты производный тип должен унаследовать от порождающего типа, достаточно прост. Он может унаследовать их все. Однако повышенная надежность во время компиляции, достигаемая при использовании именной эквивалентности, делает вопрос наследования атрибутов для производных типов более трудным. Нет причин, по которым производный тип не мог бы наследовать полное множество значений и литеральные обозначения от порождающего типа, но есть веские основания считать, что он не должен наследовать все его операции.

Рассмотрим простую программу для вычисления площади прямоугольника.

```
program площадь_ прямоугольника _
 type длина = вещественный;
   площадь = вещественный;
  var x, y : длина;
       a :площадь;
```

```
begin
  читать(x, y);
  a:=x*y;
  писать(a)
end
```

Если операция * наследуется от типа *вещественный,* то разумно предположить, что тип результата операции *x * y* таков же, как и тип операндов, т. е. в нашем случае тип *длина.* Следовательно, оператор присваивания

a:=*x***y*

незаконен! Разумеется, обойти эту проблему можно, применив функцию преобразования

a := *площадь*(*x***y*)

но это громоздко и бьет мимо цели, так как назначение производных типов состоит в запрещении некорректных операций, а не в ограничении корректных. Оператор

x := *x***y*

например, законен, но является бессмыслицей с точки зрения размерности.

Отсюда следует сделать заключение, что при полной реализации высокой надежности за счет использования производных типов язык должен обладать механизмом переопределения существующих операций и, по-видимому, также механизмом определения новых операций. Заметим, что для правильного понимания этого рассуждения нужно ясно сознавать, что, скажем, операция +, определенная для сложения двух операндов типа *a,* отличается от операции +, определенной для сложения двух операндов типа *b.* В таких случаях говорят, что операция перегружена. Какая операция имеется в виду в конкретном контексте, определяется компилятором при исследовании типов предполагаемого результата. Этот аспект будет рассмотрен далее в гл. 4.

Вернемся к вопросу наследования операций. Всегда справедливо, что операция присваивания := и операции отношения = и < > (равно и не равно) корректны для любого типа. Однако очень часто не все другие операции, определенные для порождающего типа, будут применимы для производного типа. В приведенном выше примере тип *длина* должен унаследовать от своего порождающего типа операции + и -, но не * или /. Таким образом, нужно обеспечить возможность, с помощью которой множество операций, наследуемое от порождающего типа, может ограничиваться и затем расширяться за счет определенных пользователем операций.

2.2.6. Ограничения

Из приведенного в предыдущем разделе рассуждения о наследовании атрибутов вытекает, что для поддержания логической корректности некоторых производных типов данных может потребоваться ограничить множество операций, наследуемых от порождающего типа. Хотя это и не необходимо, на практике, как правило, ограничивают также и наследуемое множество значений.

Предположим, что нужно просуммировать 100 целых значений, хранящихся в массиве *a*[*i*] (1 < = *i* < = 100). Это можно сделать с помощью простого цикла

сум :=0
for *i* := 1 to 100 **do** *сум*:= *сум* + *a*[*i*];

Но если тип *i индекс,* где

type *индекс* = *целый*;

то область значений, наследуемых объектом типа *индекс,* несомненно, без необходимости велика. На самом деле требуются только значения от 1 до 100. В общем случае максимальная надежность достигается при ограничении для объекта его области значений в точности до требуемой.

Более того, имеются серьезные причины, почему лучше ограничивать диапазон значений для типа *индекс.* Наряду с надежностью, программы реального времени должны также характеризоваться эффективностью выполнения. В приведенном выше примере каждый раз при вычислении *a*[*i*] приходится выполнять контрольный код, проверяющий истинность следующих двух условий:

i<=100 и *i*>=1

перед выбором элемента массива. Выполнение этого контрольного кода связано с дополнительными расходами, которые могут даже превосходить издержки, связанные с выполнением самой операции индексирования. Конечно, можно сказать, что проверочный код вставлять совсем не обязательно. В нашем простом случае обойтись без него можно и вполне допустимо, но в общем случае отсутствие проверки границ массива опасно. Менее трудоемкую и столь же надежную проверку границ диапазона можно реализовать, если тип *i* ограничить значениями от 1 до 100, включив ограничение диапазона в описание производного типа

type *индекс* = *целый* **range** 1.. 100**;**

Теперь тип *индекс* определяет множество целых значений, лежащих в области от 1 до 100. Снабженный этой информацией

компилятор может теперь свободно опустить проверочный код при индексации массива, вместо этого проверочный код должен применяться в тех точках программы, где происходит обновление *i*. В нашем случае *i* обновляется только в цикле **for**, у которого фиксированные границы, и, следовательно, никакого контрольного кода не нужно вовсе.

Безусловно, наш пример очень прост, и достаточно тонкий компилятор мог бы определить принадлежность *i* требуемому диапазону и без внесения ограничения диапазона в описание типа *индекс.* Но рассмотрим случай, где статическое определение диапазона *i* невозможно.

```
while not готово do
begin
  i := i + 1;
  сум := сум + a[i];
  перевычислить(готово)
 end;
```

Здесь *готово* логическая переменная, принимающая значение истина в соответствии с некоторым неизвестным условием. Если в этом случае *i* неограниченное целое, то должны применяться обе проверки границ массива. Впрочем, если величина *i* ограничена, то эти проверки передаются присваиванию обновления, но здесь нужно проверять только одно условие, а именно

$(i + 1) <= 100$

так как прибавление положительного числа к *i* не может вывести его за пределы нижней границы диапазона. Никакой статический анализ, производимый компилятором в первом случае, не может установить того, что проверка нижней границы массива не необходима, так как компилятор не располагает какими-либо сведениями о значении *i* при входе в цикл. Эта априорная информация гарантируется во втором случае ограниченным диапазоном типа *i.*

Подводя итог, отметим, что ограничение множества значений, наследуемого производным типом, до множества лишь необходимых значений увеличивает ясность и надежность программы и очень часто снижает также величину издержек работы программы, возникающих в связи с внесением необходимого проверочного кода.

И наконец, в предыдущем разделе обсуждалось требование возможности ограничения множества операций и множества значений у наследуемых производным типом. Оно может быть осуществлено при предоставлении возможности перечислить операции, которые должны наследоваться, в описании типа.

Например, используемый ранее тип *длина* мог бы быть описан так

type *длина = вещественный* **inherits +, —, <, >, < =, > =;**

Предполагается, что := , = и < > наследуются всегда. При опущении части **inherits** описания типа наследуются все операции порождающего типа. Операции, которые не наследуются производным типом, но требуются в той или иной форме, должны быть определены самим программистом (см. гл. 4).

2.2.7. Подтипы

В предыдущем разделе обсуждались преимущества использования производных типов и ограничения, налагаемые на производные типы. Преимущества ограниченного множества значений велики, и ими следует широко пользоваться при разработке программы. Однако не всегда может быть удобным или даже логически целесообразным заявлять новый тип только лишь для введения ограничений. Часто некоторые объекты из заданного класса могут быть с выгодой ограничены без изменения их статуса как членов этого класса. Это может быть сделано с помощью понятия подтипа. Например, описание

subtype *малцел = целое* **range — 128.. 127;**

определяет подтип *малцел.* Объекты типа *малцел* могут свободно смешиваться с объектами типа *целый,* не требуя явного применения преобразования типа. Конечно, присваивания подтипу могут потребовать выполнения проверки диапазона во время работы программы так же, как и для ограниченных производных типов. Очень часто, однако, этих проверок диапазона можно избежать за счет статического анализа текста программы. Описание подтипа можно также использовать для переименования типа. Например,

subtype *новцел = целый*;

дает всем объектам, имеющим тип *новцел,* те же свойства, что и объектам типа *целый,* так как спецификаций ограничения диапазона нет. Определенные таким образом подтипы имеют в точности те же свойства, что и производные типы, описанные в языке, который использует структурную, а не именную эквивалентность.

И наконец, стоит отметить, что с точки зрения программиста единственное различие между производным типом и подтипом состоит в том, что первый вводит совершенно новый класс объектов, отличный от всех остальных, в то время как последний просто накладывает ограничения на существующий. Напро-

тив, реализация производных типов и подтипов может оказаться совершенно различной. Причина этого заключается в том, что всегда безопасно представлять производный тип на базовом машинном оборудовании самым эффективным образом. Например, для хранения значения производного типа достаточно отводить слова памяти минимального требуемого размера. И это не зависит от размера слова, требуемого для порождающего типа. Подтипы же могут потребовать такой реализации, которая бы учитывала вид реализации порождающего типа. По существу, это зависит от возможности передачи подтипа процедуре в виде вызываемого по ссылке параметра. Сама процедура будет работать в предположении, что объект — порождающего типа, а не подтипа, и поэтому представления порождающего типа и подтипа должны быть совместимы.

2.2.8. Анонимные типы и подтипы

Очень часто встречаются ситуации, когда введение имени нового типа или подтипа в действительности не необходимо, так как будут описаны лишь один или два объекта этого типа. В некоторых языках допускается возможность обойтись без употребления имен типов, записывая определение типа в любом месте, где требуется имя типа. Например, перечислимый тип (см. разд. 2.3) мог бы быть описан так:

type *ключ* = (*вкл, выкл*);

var *p, q* : *ключ*;

или более просто

var *p, q*: (*вкл, выкл*);

Используя структурную эквивалентность типов, описания анонимных типов можно обрабатывать теми же самыми методами, которые применяются к описаниям обычных типов, так как алгоритмы сравнения типов касаются только структуры типа, а не его имени. В общем случае описание

var *x, y* : *T*;

обычно рассматривается как аббревиатура записи

var *x* : *T*;
y: *T*;

Но в случае использования именной эквивалентности дело обстоит не так. Безусловно, что в приведенном определении для

p и *q* предполагается, что типы обоих объектов одинаковы. В случае же, когда *p* и *q* описаны раздельно, как

var *p* : (*вкл, выкл*);
q : (*вкл, выкл*);

нет никаких оснований предполагать, что *p* и *q* одного и того же типа. На самом деле, *p* и *q* должны рассматриваться как уникальные примеры некоторого уникального анонимного типа, несовместимые ни с каким другим объектом. В частности, оператор присваивания

p := *q*

должен считаться незаконным. Более того, нет никакой функции преобразования, которую можно было бы применить для реализации этого оператора.

Отсюда следует, что именная эквивалентность резко ограничивает практическую значимость описаний анонимного типа и заставляет программиста вводить имена новых типов в ситуациях, где без них вполне можно обойтись.

Еще одна проблема, касающаяся именной эквивалентности, состоит в том, что разрешение описаний анонимного типа приводит к двусмысленности таких описаний, как

var *x* : *целый* **range** 1 . . 10;
y : *целый* **range** 1 . . 10;

потому что специфицируемые для *x* и *y* типы могут интерпретироваться либо как производные типы с ограничениями, либо как подтипы с ограничениями. Эта неоднозначность и ряд других отмеченных выше проблем лучше всего разрешаются путем простого запрещения описаний анонимного типа, так чтобы все рассмотренные примеры всегда считались описаниями подтипа. В связи с запрещением в языке анонимных типов от программиста часто будет требоваться писать дополнительные команды, но это отнюдь не плохо, так как ведет к повышению удобочитаемости программы. И не будем забывать, что программа пишется всего лишь раз, а будет читаться, возможно, многократно.

2.3. ПЕРЕЧИСЛИМЫЕ ТИПЫ

Тип определяет множество значений и множество операций. Отсюда следует, что простейший способ введения в программу типа состоит в простом перечислении одного за другим значений типа. Множеством применимых к такому типу операций будет множество, состоящее из присваивания (: =) и отношений равенства (=, < >). Если далее предположить, что значения типа упорядочены, то естественно расширить это базовое мно-

жество операций за счет операций отношений сравнения (<, >, >=, <=) и функций предшествования и следования, дающих соответственно предыдущее младшее и следующее старшее значения типа.

В данном разделе обсуждаются механизмы перечислимого типа как в отношении определенных пользователем перечислимых типов, так и в отношении предопределенных перечислимых типов.

2.3.1. Определенные пользователем перечиспимые типы

Определенный пользователем перечислимый тип может быть описан с помощью простого перечисления значений, обозначаемых этим типом. Например, предположим, что в программе требуется тип для обозначения дней недели. Такой тип можно описать следующим образом

type *дни_недели* = (*пон, вт, среда, чет, пят, суб, вос*);

Каждое значение типа обозначается именем идентификатора, и на такое значение затем ссылаются с помощью этого имени. Применяемые к объектам типа *дни_недели* операции те же, что и описанные выше. Например

вт < *среда* = истина — отношение «меньше чем»
след(*пон*) = *вт* — функция следования
пред(*суб*) = *пят* — функция предшествования

Имея тип *дни_недели,* можно определить подтипы, ограничивая область значений, разрешаемых для некоторых объектов этого типа, например

subtype *раб_дни* = *дни_недели* **range** *пон. .пят*;

Объекты этого подтипа могут принимать лишь одно из пяти значений от *пон* до *пят.* Аналогично можно ввести также и производные типы, но они, по-видимому, окажутся менее полезными, так как по самой своей природе перечислимые типы определяют объекты простого логического класса.

Перечислимые типы представляют очень мощный языковый механизм, являющийся одновременно крайне простым в принципе. Поэтому может показаться удивительным, что они появились в языках программирования сравнительно недавно. Одна из причин этого состоит в том, что присущая им надежность может быть полностью реализована только в языках программирования строгой типизации. В прежних языках функции перечислимых типов выполнялись целыми типами и механизмом

описания констант. Так, например, дни недели можно было бы описать следующим образом:

const *пон* = 1**;** *пят* = 5**;**
вт = 2; *суб* = 6;
среда = 3; *вое* = 7;
чет = 4;
type *дни_недели = целый* **range** *пон...вос*;

теперь тип *дни_недели* представлен производным целым типом, множество значений которого ограничено значениями от 1 до 7. Неудобства этого подхода состоят в непривлекательности подробного выписывания, а главное в том, что здесь отсутствует надежность, присущая настоящему перечислимому типу. Предположим, что тип *месяц_года* был аналогично определен целыми в диапазоне от 1 до 12. Тогда имея

var *месяц* : *месяц_года*;
день: *дни_недели*;

компилятор не может обнаружить ошибку в

месяц := *среда* хотя он может обнаружить такую ошибку, как
месяц :=*день*

Это происходит потому, что целая константа *среда* обозначает только целое значение. Эту ошибку можно было бы обнаружить лишь тогда, когда описание констант для *пон, вт, среда* и т. д. было бы как-то связано с типом *дни-недели.* При наличии такого связывающего механизма незачем было бы вводить базовое целое представление, которое ведет к уже описанному механизму перечислимого типа.

Несмотря на их кажущуюся простоту, имеется ряд проблем, возникающих при разработке механизмов перечислимого типа. Эти проблемы связаны с самой природой перечислимого типа и особенно остро проявляются при использовании структурной эквивалентности типа. Рассмотрим следующее описание:

var *x* : (*муж, жен*);
y: (*муж, жен*);

В предположении структурной эквивалентности типа объекты *x* и *y*, очевидно, одного типа, но описание

var *s* : (*муж, жен*);
t : (*жен, муж*);

должно считаться незаконным, так как идентификаторы *муж* и *жен* не уникальны. Впрочем, компилятору не легко обнаружить

эту ошибку. Он не может просто считать конструкцию (*муж, жен*) описанием двух идентификаторов, так как ему придется тогда отвергнуть законное описание *y,* введенное ранее. Именная эквивалентность обеспечивает решение этой проблемы, связывая перечисленные идентификаторы с уникальным именем типа. У принудительного связывания имеется дополнительное преимущество, заключающееся в том, что оно позволяет также идентификаторам перечислимого типа быть перегруженными, т. е. они могут встречаться более чем в одном описании перечислимого типа. Например,

type *цвет* = (*красный, зеленый, желтый, синий*);
предупред_статус = (*красный, зеленый*);

В большинстве случаев по контексту можно определить, какое перечисление обозначается перегруженными идентификаторами *красный* и *зеленый.* Однако в некоторых ситуациях тип придется заявить явно с помощью некоторого рода квалификаторов, например *цвет* (*красный*).

При разработке любого современного языка программирования перечислимые типы считаются одним из существенных элементов языка. Они делают программу более ясной, требуя от программиста использования имен для каждого значения типа вместо употребления безличного кода целых чисел. Они увеличивают надежность, разрешая определять типы с помощью собственного персонального множества литеральных обозначений. Кроме этого, они обеспечивают также очень удобную основу для описания стандартно предопределенных типов логический и символьный, что мы обсудим в следующих двух разделах.

2.3.2. Логические типы

Логический тип обозначает логические значения *истина* и *ложь.* Он удобно описывается, например, так:

type *логический* = (*ложь, истина*);

Три стандартные логические операции применяются обычно к операндам логического типа, давая в результате логическое значение:

not — одноместное логическое дополнение
and — двуместное логическое «и»
or — двуместное логическое «или»

Операторы отношения всегда возвращают логические значения, например 3 < 5 возвращает значение *истина.* Операции отношения являются перегруженными в том смысле, что они могут применяться к многим различным типам. В частности,

они могут применяться к логическим операндам при расширении множества определенных выше логических операций. Используемая таким образом операция отношения = реализует функцию логической эквивалентности, а операция < > служит в роли операции исключающего или. Кроме того, в связи со свойством упорядоченности описания перечислимого типа отношение <= может исполнять функцию операции импликации. В следующей простой программе иллюстрируется использование этих логических операций:

```
program логический_пример;
  var p, q, r : логический;
   begin
     p : = истина; q : = ложь;
     r : = (9 < 4) < > q;    {r = ложь}
     r : = not q and p ;     {r = истина}
     r : = p < = q;          {r = ложь}
     r : = (r = q) or q;     {r = истина}
  end
```

Основное назначение логического типа состоит в реализации условий для условного оператора и оператора цикла. Приведем пример:

if *b* **then** S_1 **else** S_2 **end if**

Здесь оператор S_1 выполняется, когда логическая переменная *b* имеет значение истина; в противном случае выполняется оператор S_2.

Как языковая особенность логический тип не представляет серьезных трудностей, но следует отметить, что часто его не просто реализовывать на многих современных вычислительных машинах. Это связано с тем, что архитектура большинства вычислительных машин обладает множеством флажков состояния (нуль, отрицательный, и т. д.), фиксирующих результаты выполнения арифметических операций. Обычно эти флажки состояния доступны лишь как условия ветвления при выполнении команд. В связи с этим оператор

if *x* = *y* **then** *x* := 1 **end if**

непосредственно переводится в машинные команды следующим образом

```
сравнить x, y;
перейти, если не равно, к L1;
x := 1;
L1:
```

Впрочем, при необходимости произвести простое присваивание логической переменной

b := *x* = *y*

по-прежнему должна использоваться команда перехода

b :=*ложь*;
сравнить x, y;
перейти, если не равно, к LI;
b := *истина*;
L1 :

Любопытно, что такой небольшой набор команд позволяет непосредственно поместить флажок состояния в ячейку одной командой без предварительного преобразования.

2.3.3. Символьные типы

Большинство языков программирования содержат предопределенный символьный тип (часто обозначаемый аббревиатурой *сим*). Объекты типа *сим* могут содержать одно символьное значение, которое представляется на машинном уровне целым в области от 0 до 255. Фактическое представление каждого символа зависит от используемого набора символов (например, ASCII EBCDIG и т. д.), и эта множественность представлений создает основную трудность при разработке символьного типа для языка программирования.

Язык для приложений в реальном времени должен давать возможность программирования применительно к большому разнообразию объектных машин, что в отношении символьных типов означает возможность определения внутри языка символьного множества. Подходящим механизмом для этого является перечислимый тип.

При описании перечислимого типа, такого, например, как тип *дни-недели* в разд. 2.3.1, каждому перечисленному значению компилятором автоматически сопоставляется базовое целое представление. Для типа мощности *n* (т. е. когда в типе описывается *n* значений) каждому значению сопоставляется одно целое число из множества 0, 1, 2, 3,..., *n* - 1. Это базовое представление называется порядковым значением перечисления и может быть определено с помощью предопределенной функции *пор.* Для случая типа *дни-недели,* например,

пор(*пон*) = 0 *пор*(*вт*) = 1 и т. д.

Отсюда следует, что если в описании перечислимого типа разрешить константы символов и перечисляемые идентификаторы, то символьное множество можно определить перечислением. На-

пример, множество символов ASCII можно определить так:

```
type сим =

(nul,  soh,  stx,   etx,  eot,  enq,  ack,  bel,
bs,    ht,   lf,    vt,   ff,   cr,   so,   si,
die,   del,  dc2,   dc3,  Dc4   nak,  syn,  etb,
can,   em,   sub,   esc,  Js,   gs,   rs,   us,
' ',   '!',  ' " ', '#',  '$',  '%',  '&',  ' ' ' ',
'(',   ')',  '*',   '+',  ',',  '-',  '.',  '/',
'0',   '1',  '2',   '3',  '4',  '5',  '6',  '7',
'8',   '9',  ':',   ';',  '<',  '=',  '>',  '?',
'@'    'A',  'B',   'C',  'D',  'E',  К     'G',
'H',   'I',  'J',   'K',  'L',  'M',  'N',  'O',
'P',   'Q',  'R',   'S',  'T',  'V',  'U',  'W',
'X',   'Y',  'Z',   '[',  '\',  ']',  '↑',  '-',
'’',   'a',  'b',   'c',  'd',  'e',  'f',  'g',
'h',   'i',  'j',   'k',  'l',  'm',  'n',  'o',
'p',   'q',  'r',   's',  't',  'u',  'v',  'w',
'x',   'y',  'z',   '{',  '|',  '}',  '~',  del )
```

Здесь порядковое значение для каждого символа соответствует коду ASCII для этого символа и естественные перечисленные идентификаторы используются для обозначения символов, не выдаваемых на печать. Описание типа *сим,* как правило, будет предопределенным для каждой реализации, но может быть легко переопределено программистом для получения нового символьного множества, что позволяет разрабатывать мобильное программное обеспечение.

Еще одним преимуществом этого подхода к реализации символьных типов является возможность определения в программе проблемно-ориентированного символьного множества. Например, в следующем описании объявляется символьное множество, удобное для представления шестнадцатеричных чисел:

type *шестн_сим* = ('0', ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’,
‘8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’);

Здесь все символы от '0' до *'F'* являются перегруженными. Так как порядковые значения этого нового типа непосредственно представляют фактические шестнадцатеричные значения, программирование существенно упрощается. Чтобы убедиться в

этом, сравните следующие две альтернативы чтения символьного значения и вычисления его шестнадцатеричного значения,

(a) Используем тип *сим*

```
var c : сим;
  шестн_знач : целый range 0. . 15;
begin
  читать(c);
  if c < = '9' then шестн_знач:=пор(c) — яор('0')
              else шестн_знач := пор(c) — пор ('A') + 10
end
```

(b) используем тип *шестн_сим*;

```
var c : шестн_сим;
  шестн_знач : целый range 0. . 15;
begin
  читать(c); шестн_знач :=пор(c)
end
```

Причина, по которой вторая версия гораздо проще, заключается в том, что процедура *читать* выполняет неявное преобразование типа, отображая представление вводных символов, которое определено типом *сим, в* представление, определенное типом *шестн-сим.* Хотя это отображение связано с существенным увеличением расходов во время обработки программой операций ввода-вывода, тот факт, что процедура чтения будет системно-определенной, предоставляет возможность более эффективного кодирования (например, при использовании кода ассемблера), что приведет к более ясной и простой программе. Другим преимуществом второй версии является ее назависимость от фактически используемого символьного множества. Первая версия будет работать правильно только тогда, когда порядковые значения цифр '0' ... '9' и букв *'A'* ... *'F'* будут идти непрерывно.

И наконец, заметим, что в некоторых ситуациях тип перегруженной символьной константы должен быть указан явно путем связывания ее с именем типа. Например, функция

пор(*'A'*)

будет, как правило, возвращать порядковое значение *сим*(*'A'*), т. е. 65. Если вместо этого требуется порядковое значение *шестн_сим,* то это должно быть явно записано

пор(*шестн_сим*(‘*A*’))

что возвратит значение 10. Функция *пор* является перегруженной функцией, автоматически определяемой для каждого нового перечислимого типа.

Описанная выше система символьных типов предлагает четкое решение проблемы работы с различающимися символьными множествами и требует лишь минимального расширения до существующего механизма перечислимого типа. Разрешая переопределять предопределенный тип *сим* и разрешая создавать новые определяемые пользователем символьные множества, такие, как *шестн__сим,* мы получаем возможность разрабатывать мобильные программы с минимальными предположениями о базовом представлении символьного множества.

2.4. ЧИСЛОВЫЕ ТИПЫ

Все языки программирования общего назначения должны обеспечивать типы данных для представления числовых величин. Языки реального времени должны, кроме того, обеспечить эффективную реализацию этих представлений, точную и машинно-независимую.

Математики выделяют два различных типа чисел. Во-первых, имеются целые числа, которые представляют бесконечное множество дискретных значений; во-вторых, имеются вещественные числа, которые представляют бесконечный континуум значений.

Представление этих двух различных типов чисел на вычислительной машине ставит ряд проблем, большинство из которых связано с требованием машинной независимости и мобильности. Целые числа могут быть точно представлены на вычислительной машине, но только в конечной области. Перемещение программы с одной машины на другую с меньшим размером слова может привести к арифметическому переполнению. Вещественные числа вынуждают кроме проблемы конечного диапазона решать также проблему конечной точности. Поэтому перемещение программы с одной машины на другую может не привести к аварийной остановке, но уменьшить точность вещественных вычислений. При выполнении вычислений в реальном времени вещественные числа могут представляться либо в формате с фиксированной запятой, либо в формате с плавающей запятой. В данном разделе рассматриваются числовые типы целый, плавающий и фиксированный с точки зрения их реализации в языках программирования реального времени. В этой связи особенно важными являются вопросы, касающиеся мобильности и легкости использования, Мы их разберем несколько подробнее.

2.4.1. Целые типы

При традиционном подходе к реализации целых чисел в языке программирования предполагается, что они будут непосредственно отображаться на машинное оборудование, так что для длины слова *n* битов и любого целого *i* соотношение

$$-2^{n-1} <= i < 2^{n-1}$$

определяет максимальный диапазон значений, принимаемых *i* (при условии использования обычного двоичного дополнительного представления). Обычно лишнее значение -2^{n-1}, не имеющее положительного симметричного элемента, опускается из определяемой области, и практически рабочим диапазоном целых чисел является

$$| i | <= 2^{n-1} - 1$$

Значение $2^{n-1} - 1$ для любой реализации является, как правило, системной константой, называемой, например, *максцел.* Следовательно, нетрудно получить меру мобильности, анализируя значение *максцел* в соответствующей точке программы, чтобы удостовериться, что имеющийся и зависящий от реализации диапазон достаточен для текущих вычислений. Если это не так, то можно либо выдать предупредительное сообщение и исключить программу, либо вызвать альтернативную подпрограмму, которая выполняет те же самые вычисления, но менее эффективно, используя «многословную» арифметику.

Ясно, что этот подход менее чем удовлетворителен. Часто оказывается очень трудно построить действительно мобильную программу так, чтобы сохранить эффективность на машине с большой длиной слова, обеспечивая правильную работу программы на машине с маленькой длиной слова.

Развитие механизма типизации (см. 2.2) дает простое решение рассматриваемой проблемы мобильности. Программист, решающий эту проблему, может определить свои собственные целые типы в начале программы. Например,

type *болцел* = *целый* **range** — 2**31..2**31 — 1;
    цел = *целый* **range** — 2**15. .2**15 — 1;
    малцел = *целый* **range** —128.. 127;

Имея эти описания производных типов, компилятор обладает достаточной информацией для реализации различных размеров целых чисел, требуемых программистом на любой объектной машине независимо от ее длины слова. На машине с 32 битами все целые могут быть представлены в рамках одного слова. На машине с 16 битами для *болцел* требуется представление с

использованием двух слов, а на машине с 8 битами — двусловное представление требуется для *цел* и четырехсловное представление для *болцел.* Таким образом, мы получаем мобильную программу и сохраняем эффективность для большой машины.

Небольшая трудность возникает при использовании этой системы в связи с самим типом *целый.* При его применении в качестве базового типа он должен рассматриваться как истинный целый тип, т. е. с бесконечной областью значений, что позволило бы получать производные большие целые типы. Впрочем, при его фактическом использовании, как в примере

var *z*: *целый*;

должен предполагаться подходящий системно-определенный диапазон. На самом деле эта проблема никогда не будет докучать программисту-практику. Ее можно обойти, если язык вообще не обеспечивает никакого целого типа, а требуется, чтобы все нужные целые типы явно строились из предопределенного, но недоступного типа *целый.*

Стандартные арифметические операции, связанные с целыми числами, обычно включают

\+ сложение — вычитание
* умножение / деление

и, кроме этого, часто добавляются

** возведение в степень
mod остаток после целого деления

Семантика операций +, —, * и ** однозначна и не требует объяснений. Операции / и mod, однако, не имеют общепринятой семантики для случая, когда один из операндов отрицателен. Проблема возникает при решении вопроса о том, чему равно —3/2: —1, —2 или неопределенно, т. е. при решении вопроса, в какую сторону, к минус бесконечности или к нулю, должно производиться округление отрицательных операндов, или же отрицательные операнды вообще не должны допускаться? Очень часто ответ на этот вопрос в старых языках можно было найти, только проведя соответствующий эксперимент для конкретной реализации. На практике большинство аппаратуры округляет результат в сторону нуля, так что это может показаться лучшим выбором, в частности еще и потому, что при этом также гарантируется соответствие для вещественных чисел, т. е. равенство

остается верным и для целых. Однако рассмотрим достаточно общий случай целого деления на степень 2, когда у компилятора должна иметься возможность воспользоваться командой сдвига, а не командой деления для увеличения эффективности: проблема в том, что сдвиг округляет в направлении минус бесконечность. Для решения этой проблемы можно предложить четыре пути:

(1) Включить проверку знака и сделать корректировку для отрицательных операндов при реализации деления с помощью операции сдвига.
(2) Запретить отрицательные операнды.
(3) Ввести две различные операции, одна из которых округляет к нулю, а другая к минус бесконечности.
(4) Опустить оптимизацию с помощью операции сдвига.

Решение 1 мало соблазнительно, так как выгода в скорости от использования операций сдвига теряется за счет операций проверки и корректировки. Решение 2 неприемлемо узко, а решение 3 усложняет язык. Так что лучшим решением нужно считать, по-видимому, решение 4, т. е. предпочтительно сохранить единственную операцию деления и специфицировать округление к нулю. Компилятор, следовательно, должен всегда использовать обычные операции деления, за исключением тех случаев, когда программист явно указывает, что операнды всегда положительнее в соответствующем описании типа, например,

type *полцел* = *целый* **range** 0..32767;

Здесь оптимизация с помощью операций сдвига возможна.

Установив семантику операции /, операцию **mod** можно определить следующим соотношением:

$x = (x/y)*y + (x \textbf{ mod } y)$

где *x к y* — целые числа. Здесь также операция **mod** может быть реализована очень эффективно, когда *y* есть степень 2, с помощью операции маскирования; но при этом значение *x* не должно быть отрицательным. Эта проблема в точности совпадает с рассмотренной выше проблемой деления, и опять лучшее решение состоит в выборе наиболее чистой семантической спецификации; попытка оптимизации производится только тогда, когда программист указывает, что это возможно, используя положительные целые производные типы. Более детальное обсуждение проблем, связанных с определением целого деления, можно найти у Вихмана (1979) и Джемисона (1980).

2.4.2. Плавающие типы

Представление вещественного числа *x* в форме с плавающей запятой состоит из мантиссы m и экспоненты *e,* таких, что для *x* справедливо $x = m.r^e$, где *r* — основание, равное в зависимости от реализации 2, 10 или 16. Точность представления с плавающей запятой зависит от числа битов, отведенных для мантиссы, и главная проблема с плавающими типами состоит в слежении за этой точностью. В старых языках, таких, как Фортран, имелись одинарный тип точности и тип двойной точности. Но так как в зависимости от реализации под мантиссу может быть отведено от 24 до 48 битов, ясно, что тип двойной точности на некоторых машинах может оказаться не более точным, чем тип одинарной точности на других.

Язык реального времени должен предоставлять тип вещественных данных, который бы гарантировал некую минимальную точность для некоторой области реализаций. Здесь, как и ранее, мобильность лучше всего обеспечивается, когда программисту разрешается определять требуемую ему точность. Например, тип *мойплав* может быть описан как новый тип, производный от предопределенного типа *плавающий* следующим образом:

type *мойплав* = *плавающий* **range** —1E40.. 1E40 **precision** 10;

Точность специфицируется здесь в терминах требуемого числа десятичных знаков.

Как и для случая типа *целый,* рассмотренного в предыдущем разделе, нужно считать, что тип *плавающий* характеризуется диапазоном и точностью, превосходящими все диапазоны и точности, фактически требуемые, когда он используется при построении новых типов. Впрочем, когда некоторый объект фактически описывается типом *плавающий,* например как в

var *x*: *плавающий*;

то используются и некоторые системно-определенные диапазон и точность, и это делает программу, использующую предопределенный тип *плавающий* непосредственно, потенциально немобильной.

С плавающим типом связывается обычное множество арифметических операций:

+ сложение — вычитание
* умножение / деление
** возведение в степень

Никаких существенных трудностей с семантикой этих операций нет, но имеет смысл рассмотреть их эффект в програм-

мах, которые требуют чисел в форме с плавающей запятой с диапазонами различающихся точностей. Предположим, что имеются следующие описания:

type *двойной = плавающий* **range** min..*max* **precision** 12;
простой = плавающий range min..max **precision** 6;
var *dx, dy* : *двойной*;
sx, sy : *простой*;

Так как тип объектов одинарной точности отличается от типа объектов двойной точности, нет неопределенности относительно типа (и, следовательно, точности) промежуточных результатов в арифметическом выражении. Например, выражение

sx + dx

незаконно, нужно явно указать функцию преобразования типа, например

sx + простой(*dx*) или *двойной*(*sx*) + *dx*

В каждом случае точность результата ясна: одинарный тип в первом случае и двойной во втором.

Следует подчеркнуть, что спецификации точности являются для компилятора ограничением только снизу, ему не возбраняется представлять все указанные выше объекты одинаково при условии, что обеспечивается точность по крайней мере 12 знаков. Спецификации точности служат для двух целей. Во-первых, программист может быть уверен, что заданная в спецификации точность будет гарантироваться для любой реализации, и во-вторых, они позволяют компилятору оптимизировать отображение определенных пользователем плавающих типов на имеющееся машинное оборудование.

В связи со спецификацией индивидуального типа для каждой точности возникает проблема чрезмерно большого числа явных употреблений преобразования типа в смешанных выражениях. В этом случае можно поступать и по-другому: считать, что ограничения точности имеют тот же статус, что и ограничения диапазона, и поэтому специфицировать их с помощью описания подтипа. Таким образом, в предыдущем примере одинарный тип мог бы быть описан как подтип

subtype *одинарный = двойной* **precision** 6**;**

В этом случае явно специфицировать преобразование типа не нужно, и *sx + dx* является законным выражением. Впрочем, семантика операций с плавающей запятой становится теперь менее ясной, так как изменения точности уже не указываются явно, а осуществляются автоматически компилятором. Более того, использование подтипов часто будет приводить к получению

менее эффективной программы, чем было бы возможно при использовании производных типов. Это связано с тем, что производные типы различны и могут быть отображены непосредственно на имеющееся машинное оборудование, в то время как эквивалентные подтипы непосредственно отображены быть не могут. Чтобы показать это, предположим, что точности, специфицируемые для типов *одинарный* и *двойной,* отображаются непосредственно на имеющееся машинное оборудование соответственно на одно слово и двойное слово арифметической аппаратуры. Объект типа *простой* всегда может храниться в форме одного слова, однако объект подтипа *простой,* возможно, потребуется хранить в форме двойного слова, например, если он передается по ссылке в процедуру, которая ожидает параметр типа *двойной.*

Итак, надежный, достаточно мобильный и эффективный плавающий тип данных может быть реализован в языке программирования с помощью достаточно гибкого механизма типизации. Различающиеся производные типы, соответствующие различным точностям, позволяют реализовать семантику и выполнение операций с плавающей запятой относительно просто. Однако использование большого числа различных типов требует частого явного употребления преобразования типа в выражениях со смешанными точностями, что приводит к более длинным и менее удобным для чтения программам. Эта проблема устраняется при использовании подтипа, что, однако, увеличивает сложность компилятора и делает менее ясной семантику операций с плавающей запятой.

И наконец, следует отметить, что в любом языке, допускающем непосредственное отображение операций с плавающей запятой на имеющееся машинное оборудование, невозможно требовать полной мобильности, так как некоторые свойства плавающих типов нельзя специфицировать семантикой языка. В частности, выбранное в качестве основания счисления значение может повлиять на точность вычислений, а это значение определяется, как правило, машинным оборудованием.

2.4.3. Фиксированные типы

Включение арифметики с фиксированной запятой является особой чертой языка программирования реального времени. Арифметика с фиксированной запятой рассматривается в общем случае как быстрый и дешевый способ выполнять практические вычисления на машинах, не оснащенных аппаратурой обработки плавающей запятой. Вследствие этого часто говорят, что возможность работы с фиксированной запятой является излишней особенностью разработки языка, так как аппа-

ратура обработки плавающей запятой скоро будет настолько дешевой, что станет стандартным свойством всех машин. Возможно, что когда-нибудь это будет и так. Тем не менее имеется большое количество производимых и используемых на практике машин, которые не обладают такой аппаратурой, и поэтому необходимость в фиксированной запятой будет актуальной еще в течение некоторого времени. Более того, есть достаточно серьезные причины, почему в некоторых случаях фиксированную запятую следует предпочесть плавающей запятой. Во-первых, данные в форме с плавающей запятой требуют больше памяти, и поэтому при хранении больших объемов данных фиксированная запятая может оказаться весьма удобной. Во-вторых, большое количество исходных данных в системах реального времени уже задается в форме с фиксированной запятой, например входные и выходные данные цифро-аналоговых и аналого-цифровых преобразователей. Поэтому, чтобы использовать плавающую запятую, нужно выполнить преобразования сначала из фиксированной в плавающую запятую, а затем - из плавающей в фиксированную. Каждое преобразование связано с потенциальной потерей точности и с некоторым дополнительным временем выполнения, так что фиксированная запятая вполне может обеспечить более простое и более точное решение. В-третьих, для любой технологии машинного оборудования операция с фиксированной запятой всегда будет быстрее операции с плавающей запятой. Поэтому некоторые специализированные применения, такие, как высокоскоростная обработка цифровых сигналов, всегда будут нуждаться в возможности работы с фиксированной запятой. И наконец, фиксированная запятая потенциально более надежна, чем плавающая. На диапазон значений, принимаемых всеми объектами с фиксированной запятой, должны быть наложены жесткие границы, предохраняющие от переполнения и потери точности, а также позволяющие выполнять эффективный анализ ошибок. Здесь возникает побочный эффект создания, вообще говоря, более надежной программы, так как компилятор и система пропуска программы могут обнаружить выход за границы диапазона, что вполне могло бы остаться незамеченным в варианте с плавающей запятой.

Хотя все приведенные выше соображения представляют сильный довод в пользу включения в язык программирования реального времени возможности работы с фиксированной запятой, следует признать, что с ней гораздо труднее работать, чем с плавающей запятой. Поэтому очень важно, что при включении операций с фиксированной запятой их следует тщательно продумать, так чтобы они были просты для понимания и использования и имели бы четкую семантику. И на самом деле,

обеспечение средств фиксированной запятой в некоторых ранних разработках языков перестало удовлетворять практиков, так как эти средства нельзя было понять должным образом и ими трудно было пользоваться.

Есть два фундаментально различных подхода к разработке фиксированного типа данных. Мы имеем в виду точное и приближенное представления.

При использовании точного представления значения представляются целыми числами в некотором специфицированном масштабе. Например, в случае

var *a, b* : *фиксированный* **range 0...10 scale** 1/8;
c : *фиксированный* **range 0...20 scale** 1/3;

а и *b* представляются целыми числами в диапазоне от 0 до 80, обозначая значения

0 1/8 1/4 3/8... 10

а *c* представляется целыми числами в диапазоне от 0 до 60, обозначая значения

0 1/3 2/3 1...20

Возникающие при точном представлении проблемы связаны с реализацией арифметических операций. Посмотрим, что нужно сделать, чтобы вычислить

$a := b + c$

Две величины *b* и *c* не могут быть сложены непосредственно, так как у них разные масштабы. Сначала нужно так преобразовать масштабы, чтобы они стали одинаковыми, т. е. вычислить их наибольший общий делитель (нод), который в нашем случае равен 1/24. Затем результат сложения должен быть приведен опять к масштабу *a*. Все эти операции масштабирования требуют умножений и делений, что связано с серьезными временными издержками во время работы программы. Более того, автоматическое приведение масштабов с помощью алгоритма нод не позволяет программисту предвидеть возможность переполнения во время выполнения арифметической операции. Проблемы масштабирования становятся особенно острыми при рассмотрении операций умножения и деления, причем это возникает даже тогда, когда все операнды имеют одинаковый масштаб.

Таким образом, реализация фиксированной запятой с точным представлением требует большого числа дорогих операций преобразования во время работы программы, а точное и простое описание семантики очень трудно. И на самом деле альтернатива использования приближенного представления гораздо

лучше соответствует нуждам программиста в реальном времени. Оставшаяся часть этого раздела будет посвящена второму подходу.

При использовании приближенного представления вещественное число представляется двоичным дополнительным кодом из $n + m + 1$ битов с весами

$$sb_n b_{n-1} \bullet \bullet \bullet\ b_1 . f_1 f_2 \bullet \bullet \bullet fm$$

где $b_i = 2^{i-1}$, $s_i = 2^{-i}$ и s — бит знака. Если m = 0, то число целое; если $n = 0$, то число - чистая дробь. В качестве примера общего случая рассмотрим представление числа 2,5 числом из 8 битов, 3 из которых дробные:

0 0 0 1 0. 1 0 0

В данном случае представление точное, но число 2,4, например, может быть представлено лишь приближенно в виде

0 0 0 1 0. 0 1 1

На практике сама двоичная точка не существенна для базового машинного оборудования, которое реализует чисто целую арифметику. Но она тем не менее важна для компилятора, который по-прежнему должен выполнять операции масштабирования перед сложением двух чисел, заданных с разной точностью. Впрочем, в отличие от случая точного представления, эти операции масштабирования могут быть выполнены простыми командами арифметического сдвига, и поэтому их реализация проста и эффективна.

Прежде чем приступить к формулированию арифметики фиксированной запятой в терминах типов и операций языка высокого уровня, имеет смысл рассмотреть базовые операции двоичной аппаратуры. При умножении двух двоичных слов получается результат длиной в два слова. Для стандартных целых типов этот результат, хранящийся в двух словах, немедленно округляется до длины в одно слово путем отбрасывания слова более высокого порядка. Если такое оборудование нужно использовать для вещественной арифметики, то полный результат длиной в два слова должен быть сохранен по крайней мере временно, чтобы можно было выполнить масштабирование перед округлением и переходом обратно к длине в одно слово. Покажем это на простом примере.

Предположим, что нужно вычислить падение напряжения на сопротивлении по измеренной величине проходящего через сопротивление тока. Известно, что максимальное значение сопротивления равно 100 Ом, а максимальное значение силы тока 10 А. Ясно, что прямое представление сопротивления и силы тока в терминах целых значений даст плохую точность.

При масштабировании чисел точность увеличится. В предположении, что используется 16-битовая арифметика, значения сопротивления можно хранить X256, а значения силы тока X2048. Это эквивалентно хранению значений сопротивления с 8 дробными битами точности, а значения силы тока с 11 дробными битами точности. Например, сопротивление 80 Ом будет храниться в виде целого числа 20480, а сила тока в 7,9 А будет храниться в виде 16 179. Отсюда, если падение напряжения на сопротивлении вычисляется умножением величины сопротивления на величину тока, получается результат 331345920, для хранения которого требуется слово двойной длины, Перед приведением результата опять к форме хранения в одном слове он должен быть перемасштабирован. Максимально возможное напряжение составляет 1000В, поэтому для напряжения удобной шкалой является X32. Таким образом, полученный выше результат нужно разделить на (2048X256)/32, т. е. на 16384, что дает значение 20223, являющееся представлением 632В. Важным здесь является то, что произведение, требующее для хранения два слова, перед округлением до размера одного слова должно быть промасштабировано.

Приведенный выше пример показывает, как должна быть реализована вещественная арифметика на уровне машинного кода. Она требует как доступа к промежуточным результатам в форме двойного слова, так и знания о длине слова базового машинного оборудования. Впрочем, на уровне языка программирования реального времени, разработка которого нацелена на мобильность, ни то, ни другое не должно касаться программиста. Требуется более абстрактная формулировка, которая позволила бы программисту пользоваться арифметикой с фиксированной запятой без учета аппаратной реализации.

Особенно простым способом формулировки приближенных операций с фиксированной запятой в языке программирования является запрещение использовать произвольную точность. Вместо этого предоставляется только один дробный тип, такой, что для любого *x,* описанного в виде

var *x* : *дроб*;

справедливо, что —1.0 <= *x* < 1.0. Использование этого подхода опирается на принцип программирования, согласно которому все вещественные значения представляются в виде дроби их максимального значения. Не требуется никакого масштабирования произведений двойной длины, так как преобразование к простой длине требует только отбрасывания слова низшего порядка. На практике операция масштабирования по-прежнему необходима для сохранения точности, но знания длины базового слова не требуется, так как максимальное значение лю-

бой дробной переменной всегда равно 1.0 независимо от того, сколько битов используется для ее представления. Наряду с операцией масштабирования полезно также разрешать объединять дробные и целые типы. Это означает, что будет три формы двойной длины:

1) целый*целый -> целый двойной длины
2) дробный*дробный -> дробный двойной длины
3) целый*дробный -> целое старшее слово, дробное младшее слово

Первая форма округляется до простой длины отбрасыванием старшего слова, вторая форма округляется отбрасыванием младшего слова. Что касается третьей формы, то подходящее округление зависит от контекста. При присваивании целому отбрасывается младшее слово, при присваивании дроби отбрасывается старшее слово. Округления внутри выражения могут потребовать явного обозначения с помощью преобразования типа.

Для пояснения сказанного рассмотрим пример. Предположим, что требуется написать программу, реализующую фильтр нижних частот 3-его порядка по Чебышеву, описываемый разностным уравнением

$$y_n = 0.011956*(x_n + 3\,x_{n-1} + 3x_{n-2} + x_{n-3}) + 1.9749\,y_{n-1} - \\ - 1.5243\,y_{n-2} + 0.4538\,y_{n-3}$$

где x_n — выборочные входные данные, полученные от аналогоцифрового преобразователя, а y_n — вычисленные выходные данные, которые должны поступить на вход цифро-аналогового преобразователя.

Общий вид соответствующей программы имеет вид

```
 1 var xn,xn1,xn2,xn3 : дроб;
 2      yn,yn1,yn2,yn3,z:дроб;
 3 begin
 4   repeat
 5      читатьацп(xn);
 6      xn:=xn! — 3;
 7      z := (0.011956B6*(xn + дроб(3*( xn1 + xn2)) + xn3))! -5;
 8      yn:=(z+ 1.9749B - 1 *yn1 - 1.5243B - 1 *yn2 +
                                          + 0.4538B - 1*yn3)!1;
 9 писатьцап(yn! 1);
10      yn3:= yn2; yn2:= yn1; yn1:= yn;
11      xn3 := xn2;  xn2 := xn1 ;  xn1:= xn
12   until всегда
13 end
```

Предполагается, что процедура *читатьацп* читает значение из *ацп* и присваивает значение в виде дроби *хп.* Процедура *писатьцап* выполняет операцию преобразования. Операция ! есть операция сдвига; первый операнд сдвигается на число битов, указанное во втором операнде, причем сдвиг осуществляется влево при положительном значении и вправо — при отрицательном. Мы считаем, что эта операция может применяться как к простым операндам, так и к операндам двойной длины. Округление результата двойной длины происходит лишь тогда, когда он становится операндом операции, требующей простого операнда (например, при сложении и присваивании). Обозначение для дробных операндов такое:

вещественноечислоВ масштаб

Например, 0.011956*В*6 эквивалентно $0.011956 * 2^6$.

Объясним, как мы строили эту программу. Рассмотрим сначала вычисление выражения

$$0.011956\ (x_n + 3\ x_{n-1}\ ,+ 3\ x_{n-2} + x_{n-3}) \qquad (2.1)$$

Сумма четырех слагаемых в круглых скобках не должна превышать 1.0. В худшем случае

$$x_n + 3\ x_{n-1}\ ,+ 3\ x_{n-2} + x_{n-3} = 8x_n = 8$$

Поэтому значение x_n сразу же после ввода должно масштабироваться с коэффициентом -3 (строка 6). Все константы должны быть представлены с максимальной точностью, т. е. по возможности ближе к 1.0, поэтому вычисление выражения 2.1 должно быть запрограммировано так:

z :== 0.011956*В*6*(*хп* + *дроб*(3*(*хп*1 + *хп*2)) + *хп*3) (2.2)

Масштаб для *z* равняется 6 + (- 3) = 3. Обратите внимание на явное использование преобразование типа *дроб,* указывающее на необходимость взятия дробного результата. Выражение

$$1.9749\ y_{n-1} - 1 - 1.5243\ y_{n-2} + 0.4538\ y_{n-3} \qquad (2.3)$$

имеет максимальное значение 3.953, поэтому, для того чтобы сумма не превышала 1.0, требуется общее масштабирование с коэффициентом -2. Масштабирование констант с коэффициентом -1 и значений *у* с коэффициентом -1 обеспечивает требуемое масштабирование, отсюда 2.3 выражается, как в строке 8 программы. Масштаб *z* теперь отличается от масштаба остальных членов в (2.3), поэтому (2.2) нужно модифицировать, перемасштабировав его с коэффициентом 2 так, как это сделано в строке 7 программы. И наконец, когда y_n выводится и записывается в *цап,* его нужно перемасштабировать обратно из его внутреннего масштаба — 1 к масштабу 0, как сделано в

строке 9 программы. Строки 10 и 11- простые присваивания, в которых реализуется необходимое обновление предыдущей выборки.

В этом примере работы с дробями следует обратить внимание на следующие моменты

a) *Преимущества*

1) Для обеспечения максимальной точности без арифметического переполнения не требуется никакого учета базовой длины слова при разработке.

2) Программа мобильна и автоматически полностью использует любую доступную длину слова.

3) Не требуется анализа сложной семантики.

4) Дробные типы легко реализуются компилятором.

b) *Недостатки*

1) Написанные для реализации с дробями программы не похожи на программы, написанные для вычислений с плавающей запятой.

2) Программист обязан приводить все переменные к новому масштабу в области от -1.0 до +1.0, в связи с этим программа не отражает прямо алгоритм, который она реализует.

3) Без знания базовой длины слова не возможен анализ ошибок.

4) Рассмотренными в одном из предыдущих разделов преимуществами ограничений диапазона нельзя воспользоваться, так как все переменные по необходимости имеют один и тот же диапазон.

На практике дробные типы данных оказались очень полезными, главным образом потому, что они принципиально просты и легки для реализации. Недостаток, отмеченный еще в п. (3), можно легко устранить с помощью спецификаций точности, например,

var *x* : *дроб* **precision** *n*;

где *n* определяет число требуемых эквивалентных значащих десятичных цифр. Главные проблемы, связанные с дробными типами, состоят в том, что они довольно низкого уровня и не допускают параллельного выполнения, предоставляемого плавающими типами. В связи с этим их трудно использовать, а программы, написанные с применением плавающей запятой, не легко переносятся на машины с аппаратурой, рассчитанной только на фиксированную запятую.

Описанный выше тип *дроб* является специальным случаем более общего фиксированного типа. В заключение этого раздела мы дадим еще одну формулировку фиксированного типа,

очень близкую к формулировке фиксированного типа из предыдущего раздела. Фиксированный тип может быть описан в терминах предопределенного типа *фиксированный* следующим образом

type *мойфиксированный* = *фиксированный* range -20.0.. .20.0
  precision 0.01;

где точность задается теперь в терминах максимальной верхней границы абсолютной ошибки. Таким образом, в нашем примере потребуется 5 битов слева от разделяющей точки и 7 битов справа. Отсюда с учетом бита для знака всего необходимо 13 битов. Тип *мойфиксированный* может быть непосредственно отображен на любую машину, располагающую 13 и более битами. Если на данной машине имеется, например, 16 битов, то компилятор отведет оставшиеся 3 бита под дробную часть для увеличения точности. Таким образом, анализ ошибки, базирующийся на заявленном максимуме ошибки 0.01, всегда обеспечит непревышение величины верхней границы ошибки. Если на машине слово содержит менее 13 битов, то для объектов типа *мойфиксированный* компилятор будет вынужден использовать многословное представление.

Как и в случае плавающего типа, фиксированный тип нужно рассматривать как тип, имеющий диапазон и точность, превосходящие все фактически требуемые диапазоны и точности. Впрочем, в отличие от типа *плавающий* его нельзя использовать непосредственно как тип. Например, описание

var *x*: *фиксированный*;

нельзя разрешить, так как диапазон и точность для фиксированных типов всегда должны быть специфицированы.

Использование производных типов позволяет сформулировать описание множества арифметических операций таким образом, при котором не требуется явного указания операций масштабирования. Вместо этого каждый производный тип вводит функцию преобразования типа, которая в применении к значению отличающегося типа вызывает автоматическое выполнение операции масштабирования. Такое преобразование типа должно всегда применяться к результатам умножения, чтобы перемасштабирование выполнялось раньше, чем округление до длины в одно слово, и, разумеется, чтобы не нарушались правила сильной типизации. Может оказаться необходимым выполнять перемасштабирование с целью поддержания точности, но это всегда делается в связи с описанным диапазоном типа и не требует знания длины базового слова. Что касается рассмотренного типа *дробь,* то полезно допустить небольшое нарушение правил сильной типизации и разрешить целым опе-

рандам смешиваться с фиксированными операндами. Когда такое случается, тип результата всегда можно рассматривать одинаковым с типом фиксированного операнда.

В качестве примера такого подхода к арифметике с фиксированной запятой перепишем приведенную ранее программу чебышевского фильтра в терминах фиксированных типов, предполагая теперь, что входные и выходные диапазоны заключены от -2048 до 2047, т. е. что *ацк* и *цап* являются 12-битовыми устройствами.

```
1   type хфикс = фиксированный range  -16384.0.. 16383.0
                                            precision 1.0;
2        уфикс = фиксированный range  -8192.0. .8191.0
                                            precision 0.25;
3   subtype zфикс = уфикс range -200.0.. 200.0;
4   var xn,xn1 ,xn2,xn3 : хфикс;
5       yn,yn1,yn2,yn3: уфикс;
6                         z: кфикс;
7   begin
8     repeat
9       читатьацп(xn);
10      z := zфикс(0.011956*(xn + 3*(xn1 + xn2) + xn3));
11      yn:=z+ 1.9749*ynl — 1.5243* yn2 + 0.4538* yn3;
12      писатьцап(хфикс(yn));
13      yn3 := yn2; yn2 := yn1; yn1 := yn;
14      xn3 := xn2; xn2 := xn1; xn1 := xn;
15    until всегда
16  end
```

Разработка этой программы значительно проще, и текст полученной программы гораздо яснее, чем первое решение в терминах дробей. В тексте программы не содержится никаких операций масштабирования и фактически единственным решением при разработке, которое зависит от аспекта фиксированной запятой, является определение фиксированных типов в строках 1, 2 и 3. Большая часть работы, которая в первой версии выполнялась программистом, переложена теперь на компилятор.

Диапазон каждого фиксированного типа определяется максимальным значением, которое может принять любое выражение этого типа. Таким образом, в строке 10 выражение

xn + 3(xn1 + xn2) + xn3*

имеет максимальное значение $8x_n$, т. е. 16383. Значения могут быть только лишь целыми, поэтому точность типа *хфикс* полагается равной 1 (строка 1). В строке 11 максимальное значение выражения

*z + 1.9749**yn1 — 1.5243*yn2 + 0.4538*yn3*

равно 3.953 * y_{max} + *z,* где *z* меньше, чем 200.0. Отсюда максимальное значение для типа *уфикс* равно 8296.0. Впрочем на практике значения *y* не являются независимыми переменными, и этот максимум никогда не достигается. Но можно показать, что диапазон от —8192,0 до 8191,0 совершенно надежен. Зафиксировав область *уфикс,* мы выбираем такую точность, чтобы максимально использовать длину слова в 16 битов (строка 2). Это, конечно, не означает, что программу нужно обязательно пропускать на 16-битовой машине. В принципе компилятор может гарантировать требуемую точность на любой машине, но если известно, что программу будут реализовывать на 16-битовой машине, то безусловно глупо специфицировать большую точность, чем это необходимо, и таким образом заставлять компилятор генерировать команды для арифметики с двойной точностью.

Константы вводятся в программу без какого бы то ни было явного масштабирования. Представление для них компилятор определят в зависимости от контекста. Если константа — операнд для операции сложения или вычитания, то она представляется с той же точностью, что и второй операнд. Если она операнд для умножения или деления, то она представляется с максимально возможной точностью, такой, чтобы общее число используемых битов было таким же, как и у второго операнда. Случаи неоднозначности можно разрешить, вводя константу как аргумент функции явного преобразования типа. Поэтому в строке 10 программы константа 0.011956 будет представлена с использованием того же числа битов, что и для *хфикс,* т. е. 15.

Компилятору в общем случае потребуется ввести команду проверки диапазона перед присваиванием фиксированной переменной. Впрочем, статический анализ часто может показать, что это необязательно. Например, он может определить, что в строке 10 значение выражения

$$xn + 3*(xn1 + xn2) + xn3$$

ограничено значением *хфикс,* и поэтому максимальное значение, которое может быть присвоено *z*, есть

$$0.011956*16384.0=195.9$$

Следовательно, он спокойно может опустить проверку диапазона для этого присваивания.

И наконец, отметим, что в строках 10 и 12 требуется явное преобразование типа. Во всех этих случаях компилятор реализует преобразование с помощью простых команд арифметического сдвига.

Мы рассмотрели два подхода к разработке приближенного представления фиксированного типа в языке программирования реального времени. Первый из них базировался на чистых дробях, а второй на фиксированной запятой с произвольной точностью. Сравнение количества страниц, посвященных обсуждению фиксированных типов, с количеством страниц для других предопределенных типов указывает на разнообразие проблем, встающих при их разработке.

Подход, основывающийся на чистых дробях, очень прост для реализации и не представляет для программиста трудностей при овладении его семантикой. Однако он требует переписывания алгоритма, чтобы все переменные были чистыми дробями, и это может привести к не очень ясным программам. При подходе, основывающемся на фиксированной запятой, несколько более просто разрабатывать программу, что ведет к получению более ясного текста программы, но этот подход значительно труднее реализовывать. Компилятор должен отслеживать все внутренние преобразования масштабов и определять, где нужно вставлять команды проверки диапазона. В самой общей оценке этот подход следует предпочесть, когда допустима повышенная сложность компилятора, главным образом в связи с возможностью писать более ясные программы. Кроме этого, фиксированный тип является более точным дополнением к плавающему типу, чем дробный тип.

2.5. ТИПИЗАЦИЯ В ПРАКТИЧЕСКИХ ЯЗЫКАХ

Можно привести много примеров слабо типизируемых языков, которые обладают свойствами, описанными в разд. 2.2.1. Одним из характерных языков этого класса является CORAL 66 (Вудвард и др., 1970), который имеет систему типизации языка Алгол 60, но существенно менее надежен. CORAL обеспечивает только числовые типы, и базовый целый тип выполняет много различных задач. Наряду с представлением чисел его можно также рассматривать как битовую строку и использовать для хранения символьных значений. Следует подчеркнуть, что разделение на слабую и сильную типизацию может оказаться дезориентирующим в контексте оценки относительной надежности языка. Например, Алгол 68 (ван Вейнгаарден, 1975) в соответствии с определениями, данными в этой главе, синтаксически слабый язык, но тем не менее это относительно надежный язык. В области реального времени язык RTL/2 (Барнес, 1976 и гл. 11), базирующийся на Алголе 68, обладает большей надежностью, чем CORAL, хотя эти языки одного порядка сложности.

Идеи сильной типизации, введенные в разд. 2.2.2, впервые появились в форме практической разработки в языке Паскаль (Вирт, 1971а), где наряду с демонстрацией их преимуществ выявились и некоторые трудности. В частности, в языке никак не были явно определены правила эквивалентности типов, и оказалось, что ни структурная, ни именная эквивалентности полностью не удовлетворительны (Хаберманн, 1973; Уэлш, Снеерингер и Хоар, 1975; Тенант, 1978). Примеры лучших систем типизации, базирующиеся на структурной эквивалентности, можно найти в языках Евклид (Лампсон и др., 1977) и Jellow (SRI, 1978). Отметим также, что, хотя Паскаль в общем сильно типизируемый язык, его числовая типизация, как это ни странно, слабая в связи с тем, что она разрешает автоматическое преобразование целых типов в вещественные типы.

Используемая везде в этой главе именная эквивалентность была принята в языках Redl (Нестор, 1978) и Green (Ихбиа, 1978), а также в версии языка Ада (см. гл. 13). Языки Redl и Green одинаковы, за исключением того, что в Redl не разрешаются анонимные типы, рассмотренные в разд. 2.2.8. И наконец, нужно отметить язык Меза как пример языка, в котором удалось объединить структурную эквивалентность, именную эквивалентность и автоматическое преобразование типа в единой схеме (Гешке, Моррис и Сатертвайте, 1977).

За исключением фиксированной запятой предопределенные типы, описанные во второй части этой главы, достаточно одинаковы. Принцип связи явного диапазона с любыми численными типами развился из начальной идеи поддиапазона в языке Паскаль в детально разработанное понятие ограничения диапазона в языке Ада. Интересно отметить, впрочем, что в язык JOVIAL (Шоу, 1963), разработанный на десять лет раньше языка Паскаль, была включена возможность спецификации числового диапазона, хотя его система типизации слишком слаба для использования этой возможности в полной мере.

Фиксированная запятая всегда была дискуссионным вопросом в разработке языков программирования реального времени. CORAL 66 является примером языка, в котором возможность фиксированной запятой так не нравилась программистам, что ею фактически перестали пользоваться. С другой стороны, дробно-фиксированный подход в языке RTL/2 оказался относительно успешным. Первоначальные предписания «особой надежности» разработки потребовали метода точного представления при трактовке фиксированной запятой, это привело к тому, что разработка «особо надежного» языка оказывалась либо слишком запутанной в случае языка Blue (Softech, 1978) или недостаточной в случае языка Green. Во время периода оценки языка Ада приводились сильные доводы в пользу

того, что чистые дроби наиболее соответствуют запросам программирования в реальном времени (Фрогат, 1978), но в конечном счете в язык Ада был включен механизм, базирующийся на приближенном представлении фиксированной запятой. В качестве примера аналогичного подхода на базе структурной эквивалентности см. язык SYCOL (Янг, 1979).

Глава 3.
СТРУКТУРНЫЕ ТИПЫ ДАННЫХ
3.1. СТРУКТУРИРОВАНИЕ ДАННЫХ

Во второй главе описывались простые типы данных. Объект данных любого простого типа обладает тем свойством, что он содержит одно и только одно значение. Но для решения большинства задач с помощью программ требуется возможность описания объектов данных, которые содержат много значений. Такие объекты описываются структурными типами данных, которые необходимы по двум причинам. Во-первых, для разработки программ методом сверху вниз нужно уметь описывать данные на различных уровнях. Например, программа, предназначенная для анализа динамических деформаций в некоторой механической структуре, может иметь в качестве входных данных множество квантованных волн, полученных на измерителях нагрузки. На высшем уровне разработки входные данные будут рассматриваться концептуально как единое целое. По мере уточнения проекта внутренняя структура данных будет все более детализироваться. Сначала будут различаться волны с разных датчиков, затем каждая волна будет разбиваться на конечные сегменты и затем в каждом сегменте будут идентифицироваться отдельные выборки. Таким образом, первоначальная структура данных описывается в терминах более простых структур данных, которые в свою очередь описываются в терминах еще более простых структур данных до тех пор, пока в конце не получается описание в терминах простых объектов данных, из которых в конечном счете составлена структура.

Во-вторых, данные должны быть структурированы так, чтобы их можно было эффективно выбирать. Без структурирования данных каждому простому объекту данных должно быть поставлено в соответствие уникальное имя, и такое имя является единственным средством ссылки на данные. Поэтому если одно и то же действие должно применяться к набору одинаковых объектов данных, для обработки каждого индивидуального

объекта придется написать свой собственный оператор. При группировке объектов в структуры данных и дальнейшей ссылке на каждый объект с помощью индекса можно будет, например, использовать один-единственный оператор в цикле для выполнения требуемых операций.

Рассмотрение различных способов, какими нам бы хотелось структурировать данные, показывает, что требуются два различных механизма (Хоар, 1972). Во-первых, механизм массивов позволяет группировать набор объектов идентичного типа таким образом, что каждый объект выбирается с помощью индекса или указателя Во-вторых, механизм записей позволяет группировать набор объектов возможно самых различных размеров таким образом, что выбор каждого объекта осуществляется с помощью имени компоненты. Эти два механизма можно комбинировать при построении структур данных произвольной сложности.

Кроме механизмов массивов и записей имеется механизм множеств, обеспечивающий существенно отличающиеся возможности. При задании простого типа данных, ограниченного конечным диапазоном значений и называемого базовым типом, структуру множество можно определить как структуру, значением которой является множество значений базового типа. Множества важны потому, что они позволяют структуры данных, традиционно представляемых в программировании в реальном времени битовыми строками, описывать на высоком уровне и гораздо более надежно

В этой главе исследуются механизмы массивов, записей и множеств для структурирования данных. Затем следует обсуждение механизмов распределения памяти В частности, описываются динамически распределяемые структуры данных и обсуждаются их разработка и реализация в контексте языков программирования в реальном времени.

3.2. МАССИВЫ

3.2.1. Понятие массива

Понятие массива является, по-видимому, наиболее широко известным из всех механизмов структурирования данных и в той или иной форме имеется фактически во всех языках программирования. По существу, массив - это группировка набора данных идентичного типа. Массиву дается имя, обозначающее всю группу, и механизм индексации, позволяющий ссылаться на индивидуальные элементы группы. Так, например, описание

type *элем_век* = **array**[1.. 10] **of** *элем*;

заявляет тип *элем_век* как объединение 10 значений типа *элем* в группу, индексируемую целыми числами от 1 до 10. Теперь можно описывать объекты типа *элем-век* обычным путем, например

var *x,y* : *элем_век*;

Индивидуальные объекты этих массивов обозначаются именем массива и следующим за ним в квадратных скобках индексом, например

x[1]:=*y*[10]

Здесь последний элемент массива *y* пересылается в первый элемент массива *x*.

Разумеется, операции можно выполнять над объектами регулярного типа[2], рассматриваемыми как единое целое, не касаясь их внутренней структуры. Такие операции, как правило, будут явно определяться программистом, но операции присваивания и проверки равенства универсальны и применимы ко всем типам. Отсюда следует, что эти две операции должны быть встроены в язык так, чтобы их при реализации можно было оптимизировать компилятором. Например,

x := *y*

семантически эквивалентно

for *i*:=1 **to** 10 **do** *x*[*i*] := *y*[*i*]

но первое может быть реализовано с помощью операции блочного копирования, в то время как при реализации второго каждая компонента обеих структур должна индивидуально распаковываться и копироваться

Массив можно представить как реализацию функции отображения. В приведенном выше примере объекты *x* и *y* определяют отображение из множества целых чисел от 1 до 10 на множество значений типа *элем.* Такое отображение можно определить для любого дискретного простого типа данных, а не только для целых. Так, например, следующее описание

type *норма_оплаты* = **array**[*пон. .суб*] **of** *ежедневная_ставка*;

определяет регулярный тип *норма-оплаты,* индекс которого принадлежит перечислимому типу *дни_недели,* определенному в разд. 2.3.1.

Описанный выше механизм массивов принципиально очень прост, но в нем нет достаточной гибкости, необходимой в практических языках реального времени. Это отсутствие гибкости

---

[2] В оригинале array type — *Прим перев*

связано с тем, что границы индексов массива явно указаны в описании типа. Если предполагается работа в рамках сильной типизации, то это запрещает программировать некоторые операции. Например, довольно общей является необходимость написания процедур, которые могли бы работать с массивами переменной длины. Это становится невозможным, когда границы массива являются частью типа, так как типы формальных и фактических параметров не будут согласовываться при вызове процедуры. Кроме того, возможность создавать динамические массивы, т. е. массивы, размер которых определяется во время работы программы, очень полезна в таких применениях, как обработка текстовых строк, но ясно, что это связано с определенными трудностями, когда границы массива являются частью типа.

Более развитое и полезное определение механизма массивов требует, чтобы массивы эквивалентных типов имели эквивалентные типы индексов и компонент, но границы индексов могут быть и не эквивалентны. В следующем разделе дается такое определение на базе именной эквивалентности. Впрочем, прежде чем дать это определение, методологически полезно рассмотреть проблемы, встречающиеся при попытке сформулировать его на базе структурной эквивалентности. Рассмотрим описания

```
type век_частоты = array[1.. 100] of мой_плав;
     век_скорости = array[1.. 100] of мой_плав;
```

При структурной эквивалентности объект типа *век_частоты,* безусловно, совместим по присваиванию с объектом типа *век_скорости,* даже если они являются логически не соотносящимися объектами данных. Другими словами, при формулировке структурированных типов возникает в точности такая же проблема, как та, что была рассмотрена в предыдущей главе для простых типов. Более того, если даже уменьшение степени надежности, связанное с данными определениями структурной эквивалентности, приемлемо, нужно иметь в виду, что выполнение в компиляторе алгоритмов структурной эквивалентности может весьма значительно уменьшить скорость компиляции. Все эти трудности могут быть разрешены с помощью именной эквивалентности.

3.2.2. Типизация с помощью массивов

Из приведенного выше рассуждения вытекает, что границы индексов массива не следует считать частью типа. Однако простое игнорирование границ индексов привело бы к неприемле-

мо ненадежному языку. При именной эквивалентности типы эквивалентны лишь тогда, когда они имеют одно и то же имя. Поэтому вопрос о том, следует ли считать границы индексов частью типа, не имеет значения, когда решают вопрос, являются ли два объекта регулярного типа совместимыми по присваиванию. Предположим, что описанный выше тип *элем_век* определен следующим образом:

type *элем_век* — **array**[*целый*] **of** *элем*;

где специфицированы лишь типы индекса и компонент. Регулярный тип *элем_век* определяет теперь множество значений регулярного типа, где каждое его значение состоит из одного или более компонентных значений. Фактические объекты регулярного типа можно теперь описать, специфицировав ограничения на диапазон индексного типа, например

var *x,y* : *элем_век*[1..10];

Это в точности соответствует описанию простого типа данных с ограничениями на диапазон. Другими словами, *x* и *y* описываются как анонимные подтипы типа *элем_век.* Регулярный подтип может, разумеется, быть описан явно:

subtype *мал_век* = *элем_век*[*0*.. 9];

Объект подтипа *мал_век* может быть далее описан так:

var *z*: *мал_век*;

Объекты *x, y* и *z* все одного и того же типа и поэтому совместимы по присваиванию, например оператор *z* := *x* законен. Компилятор будет проверять, одинаково ли число компонент в каждом объекте регулярного типа, подобно тому как диапазоны простых подтипов должны проверяться во время присваивания.

Аналогично описанным присваиваниям целиком всему массиву можно ссылаться на вырезку из массива, заменяя индекс индексным диапазоном. Например,

z[6..9] := *x*[1.. 4]

присваивает первые 4 элемента *x* последним четырем элементам *z.* Здесь компилятор может проверить, что число компонент в каждой вырезке из массива при вызове одинаково. В более общем случае, например, для

z[0.. *i*] := *x*[1..*j*]

такая проверка должна производиться во время работы программы.

В эту схему легко укладываются динамические массивы при простом разрешении специфицировать ограничения на диапазон во время работы программы, например

var *гиб_объект* : *элем_век*[1.. *размер*];

где *размер* есть переменная. Конечно, реализация динамических массивов в своей основе другая, здесь выполнение проверок границ массивов при присваивании во время компиляции становится невозможным, но с точки зрения программиста они мало отличаются от статических массивов.

Описанный механизм прост и надежен. Он позволяет описывать семейства объектов регулярного типа, имеющих общие логические свойства. Правила сильной типизации в сочетании с именной эквивалентностью обеспечивают невозможность появления в одном и том же операторе присваивания регулярных объектов, имеющих одинаковую структуру, но логически различных. Требование, чтобы все объекты регулярного типа описывались с указанием подтипа или ограничений диапазона, можно обойти с помощью сокращенных обозначений. Например, текст

type *элем_век2* = аггау[1 . . 10] **of** *элем*;

представляет собой сокращение записи

type **** = **array**[*целый*] **of** *элем*;
subtype *элем_век3,* = ****[1 . . 10];

где ****является обозначением анонимного идентификатора.

3.2.3. Многомерные массивы

На компонентный тип массива не накладывается никаких ограничений, и поэтому, описывая массивы, компонентами которого являются массивы, можно определить массивы различных размерностей, например,

type *век* = **array**[*целый*] **of** *плавающий*;
матрица = **array**[*целый*] **of** *век*;

определяет тип *матрица* как двумерный массив. Впрочем, программисты обычно предпочитают более краткую запись:

type *матрица* = **array**[*целый, целый*] **of** *плавающий*;

Объекты матричного типа описываются обычным путем, при этом ограничения на индексный диапазон специфицируются по каждому измерению, например

var *imat* : *матрица*[1..3, 1..3];

3.2.4. Конструкторы массивов

Стоит заметить, что, хотя система обозначений для конструктора массивов и не является языковым элементом в узком смысле, она представляет собой ценную языковую особенность. Удобно записывать список значений, заключенный в круглые скобки, впереди которого находится имя типа. Например

iвек := *целый_вектор*(0, 1, 2, 3)

присвоит специфицированные значения последовательным компонентам *iвек,* где предполагается, что *iвек* есть массив из 4 целых значений. В случае когда компоненты массива тоже структурированы, можно применить вложенные круглые скобки, например

iмат := *матрица*((0, 0, 0), (1, 2, 3), (—1, —2, —3))

где *iмат* есть матрица целых значений размера 3X3.

В частном случае символьных массивов для построения массива более подходит строковая запись. Например

мое_имя := *'Стив'*

присваивает символьному массиву *мое_имя* начальное значение *'Стив'.*

3.2.5. Реализация массивов

За исключением динамических массивов, описанная выше система типизации массивов не связана с чрезмерными издержками времени работы программы по сравнению с любой другой сравнимой системой. Методы реализации массивов произвольной размерности и произвольного компонентного типа описываются в большинстве книг по разработке компиляторов (см., например, Грис, 1971; Ахо и Ульман, 1978). По существу, имеются два различных метода, которые можно применить, когда компоненты массива имеют структурный тип, например, другого массива. Либо компоненты могут храниться последовательно, и тогда адрес элемента массива вычисляется умножением смещения индекса на размер компоненты, либо массив можно хранить как последовательность указателей на каждую компоненту. Последний метод уменьшает число требуемых операций умножения, что важно, когда в базовой машине нет аппаратного умножения, но требует дополнительной памяти.

С точки зрения разработки языка мало что можно сделать для увеличения эффективности доступа к массиву. Некоторая небольшая экономия может быть получена при сведении нижней границы всех индексных диапазонов к 0, но этим вряд ли.

нужно заниматься, если учесть влияние такого сведения на гибкость программирования.

Аналогично можно ограничить величину размерности до единицы или двух, а типы компонент ограничить простым типом, но и такие ограничения сильно влияют на гибкость программирования и на ясность разработки программы.

Включение динамических массивов при разработке языка программирования реального времени обосновать гораздо труднее. В некоторых обстоятельствах они позволяют существенно экономить память, но, как правило, это не окупает издержки времени работы программы, неизбежные при их обеспечении в жесткой среде реального времени. Впрочем, следует заметить, что мы получаем большие выгоды, когда все программное обеспечение, задействованное в проекте, написано на одном языке. Из этого следует, что язык реального времени должен быть адекватным системному программному обеспечению, в частности своему собственному компилятору, а также целевой области применения. Поэтому такие конструкции, как динамические массивы, с полным основанием должны включаться в язык реального времени, если даже они являются отличительной особенностью работы в режиме пакетной обработки. В конце концов издержки времени работы программы возникают лишь тогда, когда реализованная особенность действительно используется.

3.3. ЗАПИСИ

3.3.1. Фиксированные записи

В отличие от механизма массивов механизм записей позволяет группировать набор объектов различающихся типов. Каждому объекту в записи дается компонентное имя, с помощью которого на него можно ссылаться, например

```
type авт = record
              марка : изготовитель;
              размер_двиг : мощность;
              цвет : цвет_авт
           end;
```

определяет тип записи, называемый *авт.* На компоненту этой записи ссылаются добавлением к имени записи имени компоненты, которые разделяются точкой. Например, если имеется

```
var мой_авт, любой_авт : авт;
```

то оператор присваивания

```
мой_авт.размер_двиг := 1000
```

зашлет в компоненту *размер_двиг* записи *мой_авт* значение 1000. Как и в случае массивов, операции присваивания и проверки на равенство, определенные для всей записи, можно с успехом ввести в язык; например,

любой_авт := *мой_авт*

присваивает всю запись *мой_авт* записи *любой_авт.*

Тип записи *авт* является примером фиксированной записи. Проблем разработки в связи с этим механизмом не возникает; он вполне надежен, хотя и следует заметить, что определение структурной эквивалентности имеет тот же недостаток в связи с проблемой надежности, что был отмечен для массивов. Когда описываются две логически различные записи, у которых случайно оказались идентичные компонентные типы, то с точки зрения структурной эквивалентности они будут одинаковы. Поэтому здесь, как и везде в этой книге, мы предполагаем именную эквивалентность.

Как и для массивов, для записей полезно иметь механизм подтипов. Предположим, например, что нужно записать информацию как об автомобилях, так и о фургонах. Запись о фургонах может быть такой:

```
type фур = record
        марка: изготовитель;
        размер_двиг: мощность;
        max_груз: вес
       end;
```

Две первые компоненты записи *фур* идентичны первым компонентам записи *авт,* но третьи компоненты различны. На практике большинство операций, применимых к автомобилям, будут также применимы и к фургонам. Неудобно иметь две записи, определенные различными типами, и поэтому предпочтительнее, если бы они могли быть определены как подтипы одного-единственного комбинированного типа [3]. Есть два различных подхода к реализации этого требования. Первый состоит в использовании вариантных записей, а второй - в введении совершенно нового типа, называемого объединением. Мы их обсудим в следующих двух разделах.

3.3.2. Вариантные записи

Вариантная запись состоит из некоторого числа фиксированных компонент, за которыми следует несколько вариантов. Каждый вариант состоит из одной или более компонент, которые

[3] В оригинале record type. — *Прим. перев.*

рассматриваются как часть записи только тогда, когда выбирается этот вариант. Например, тип средство транспорта, определяемый ниже, есть вариантная запись:

```
type вид_транс = (авт, фур);
     транс = record
                 марка : изготовитель;
                 размер_двиг : мощность;
                 case вид : вид_транс of
                    авт : (цвет : цвет_авт);
                    фур : (тах_груз : вес)
                 end;
```

Компонента *вид* называется признаком и имеет специальное назначение. Когда значением *вид* является *авт,* выбирается вариант с меткой *авт,* т. е. запись средство транспорта становится точно такой же, как и приведенная ранее запись автомобиль. Аналогично, когда значением *вид* является *фур,* выбирается вариант с меткой *фур.* Вариантную часть записи можно рассматривать как сложенные вместе две компоненты *цвет* и *тах-груз.* Кроме своей функции указания выбора текущего варианта, компонента признак трактуется как обычная фиксированная компонента записи, т. е. если задано

```
var машина: транс;
```

то оператор присваивания

```
машина.вид := авт
```

установит компоненте признаку значение *авт,* указывающее, что выбран вариант автомобиль.

Тип *транс* представляет множество двух подтипов записи. Объект типа *транс* может представлять значения каждого подтипа простым изменением вариантной части. Впрочем, если заранее известно, что объект комбинированного типа будет принимать лишь значения одного подтипа, то, описывая его явно как этот подтип, мы обеспечиваем дополнительную надежность и ясность программы, например

```
subtype вид_авт = транс(авт);
var мой_авт : вид_авт;
```

описывает объект *мой_авт* подтипа *вид_авт.* То же самое можно получить с помощью

```
var мой_авт: транс(авт);
```

В обоих случаях спецификация признакового значения *авт* является по существу ограничением варианта. Присваивание

```
машина := мой_авт
```

не требует никаких проверок во время работы программы, тогда как присваивание

мой_авт := *машина*

требует, чтобы признаковое поле объекта *машина* проверялось, так как оно должно содержать значение *авт.*

Описанный механизм обеспечивает практичное решение подтипизации записи. Впрочем, чтобы гарантировать его надежность, на использование признакового поля записи вариантного типа должно быть наложено ограничение. Предположим, что объекту комбинированного типа *машина* присваиваются такие значения:

машина.марка := *форд*;
машина.размер_двиг := 2000;
машина.вид := *авт*;
машина.цвет := *голубой*;

и затем вслед за этими присваиваниями выполняется еще одно присваивание

машина.вид := *фур*

Теперь наш объект данных *машина* несогласован. Если бы фактически последнее присваивание было первым в последовательности двух следующих:

машина.вид := *фур*;
машина.max_гр уз := 2240;

то все было бы в порядке. Если же за первым присваиванием не следует оператор, инициализирующий *max_груз,* то рассматриваемый объект данных остается несогласованным. Любой выполняемый вслед за этим оператор, такой, как

груз := *машина.max_груз*

привел бы к тому, что переменной *груз* было присвоено бессмысленное значение.

Решение данной проблемы надежности состоит в том, чтобы разрешить пользоваться признаковой компонентой записи, когда к ней происходит индивидуальный доступ, только в режиме чтения. Единственным способом модифицировать признак в этом случае является присваивание всей записи. Из этого ограничения немедленно следует, что язык должен обеспечить конструктор записи, дающий возможность инициализации вариантной записи. Здесь можно пользоваться простой нотацией, аналогичной конструктору массива, описанному в разд. 3.2.4. Например, если имеется

var *v*1,*v*2 : *транс*;

то следующие два присваивания можно использовать для инициализации этих двух объектов типа *транс*:

*v*1 := *транс*(*форд,* 2000, *авт, голубой*)
*v*2:= *транс*(*фиат,* 3000, *фур,* 2240)

здесь инициализация *v*1 происходит при выборе варианта автомобиль, а *v*2 — при выборе варианта фургон. Присваивание приводит запись *v*1 *к* варианту фургон надежным способом. Любая попытка инициализировать запись, используя несогласованный конструктор, например

*v*1 :=*транс*(*форд,* 2000, *фур, голубой*)

может быть обнаружена компилятором, и поэтому это ограничение на присваивание признаковой компоненте обеспечивает согласованность вариантных записей во все время работы. Чтобы обеспечить надежность при доступе к компоненте вариантной записи, всегда нужно выполнять проверку для подтверждения текущей выборки вариантной компоненты, на которую сделана ссылка. Например, при выполнении присваивания

груз := *v*2. *max - груз*

компилятор, как правило, должен вставлять проверочные команды, подтверждающие, что значение компоненты *вид* есть *фур.* Очень часто, однако, компилятор сможет опустить проверочные команды, анализируя контекст, в котором используется этот вариант. Например, очевидно, что в следующем операторе выбора можно обойтись во время работы программы без команд проверки:

case *v*1.*вид* **of**
авт : *v*1.*цвет* := *красный*;
фур : *v*1.*max.груз* := 2240
end

3.3.3. Объединения

Механизм вариантной записи дает средство создания подтипов записи. Понятие объединения обеспечивает программиста примерно такой же возможностью, но несколько отличным способом. В некотором смысле объединение — это более общий механизм, чем вариантные записи, так как он является самостоятельным механизмом структурирования данных.

Тип объединение обозначает объединение двух или более произвольных типов. Например,

type *транс_вариант* = **union**(*цвет_авт, вес*);

описывает *транс_вариант как* объединение двух типов *цвет_авт* и *вес.* Объект этого типа будет иметь две компоненты. Первая компонента будет признаком, а вторая - объектом одного из объединенных типов. Признак программисту совершенно недоступен и поддерживается автоматически компилятором. Он обновляется всегда, когда объекту типа объединения производится присваивание. Например, при задании

var *vv* : *транс_вариант,*

и выполнении

vv := *красный*
мы получим значение *красный* у объекта *vv* и значение признака, указывающее, что выбирается тип *цвет_авт.* Всегда при чтении значения *vv* проверяется, что значение признака согласовано. Например, отношение

vv := *красный*

законно, тогда как

vv :=1000

приведет к выдаче сообщения об ошибке.

Так же, как и в случае вариантных записей, часто бывает нужно определить текущий вид объекта типа объединение. Удобный способ выполнения этого состоит во введении оператора специального вида. Например, оператор

case *vv* **of**
цвет_авт : *vv* := красный;
вес: *vv* := 2240
end

делает то же, что и приведенный в конце предыдущего раздела оператор **case,** но метки выбора являются теперь именами типа.

При наличии указанного типа объединение вариантные записи можно построить, включив компоненты типа объедине-

ние. Например, тип средство транспорта может определяться так:

```
type транс = record
        марка : изготовитель;
        размер_двиг : мощность;
        другие_данные : вариант_транс
    end;
```

где *вариант_транс* есть определенный выше тип объединение.

3.3.4. Варианты или объединения!

Выше описаны два механизма реализации подтипов записи. В предположении, что предложенные для использования явной признаковой компоненты ограничения налагаются на механизм вариантных записей, варианты и объединения обеспечивают виртуально одинаковые уровни надежности. Впрочем имеются некоторые отличия, которые скорее относятся к их использованию, а не разработке.

Тип объединение предлагает механизм объединения нескольких типов и не ограничивается ролью механизма подтипа записей. Например, следующий фрагмент определяет «символьный» массив, позволяющий обозначать символы значениями типа *сим,* или *мал_цел,* т. е. либо в абстрактных, либо в реализационных терминах.

```
type мал_цел=целый range 0..127;
    машин_сим = union(сим, мал_цел);
    сим_массив = array[целый] of машин_сим;
```

В данном случае механизм объединения позволяет ввести более естественное определение, чем эквивалентное определение, использующее вариантные записи, т. е.

```
type представление_сим = (выс, низ);
    мал_цел = целый range 0.. 127;
    машин_сим = record
            case пред : представление_сим of
             выс : (абстрактный : сим);
             низ : (реализационный : мал_цел)
            end;
  сим_массив = array[целый] of машин_сим;
```

Здесь запись *машин_сим* действительно очень громоздка.

При реализации такой вариантной записи, как описанная выше типа средство транспорта, тип объединение в отличие от механизма явной вариантной записи является несколько более гибким. При работе с определенным в разд. 3.3.2 типом сред-

ство транспорта нельзя изменить вариантную компоненту, не изменяя всей записи, в отличие от определения, приведенного в разд. 3.3.3, которое это разрешает. Таким образом, объединение предоставляет большую гибкость, но может оказаться менее надежным и не обладает четкостью определения вариантной записи. Также очевидно, что вид транспорта в первом случае может быть прочитан непосредственно, тогда как во втором случае о нем можно судить лишь косвенно. Кроме этих небольших различий, оба механизма достаточно сходны.

3.3.5. Реализация записей

Фиксированные записи являются по существу средством, обрабатываемым во время компиляции, так как адрес любой компоненты может быть определен статически. Но вариантные записи (или объединения) сопряжены с некоторыми издержками во время работы программы, если нужно обеспечить надежность, так как она связана с необходимостью проверять признаковые компоненты перед обращением к вариантной компоненте. Во многих случаях эта проверка во время исполнения программы может быть опущена при учете контекста, в котором встречается соответствующая ссылка, и потому данный механизм не будет излишне дорогим.

Обычно каждая вариантная часть записи хранится в одних и тех же ячейках памяти. Поэтому компилятор должен отводить достаточно памяти, чтобы удовлетворить наибольшему из указанных вариантов. Это может привести к неэффективному использованию памяти в тех случаях, когда про некоторые объекты данных известно, что они принадлежат лишь подмножеству всех возможных вариантов, определенных для данного типа. Такого расточительства памяти можно избежать, если использовать явные подтипы записи, такие, как *вид_авт* из разд. 3.3.2. Когда объект описан таким подтипом, компилятор должен отвести достаточно памяти только для специфицированного варианта. Это будет особенно важно, когда различные варианты типа запись имеют сильно отличающиеся размеры.

3.4. МНОЖЕСТВА

3.4.1. Понятие множества

Множественный тип есть тип, значениями которого являются множества значений, выбранных из некоторого другого типа, называемого базовым для множественного типа. В качестве примера предположим, что в программе управления лиф-

том каждый обслуживаемый лифтом этаж обозначается одним из значений следующего типа:

type *этаж* = (*подвал, этаж*1*, этаж*2*, этаж*3*, этаж*4);

Тогда множество вызовов лифта в каждый конкретный момент может быть обозначено типом множество так:

type *множ_этаж* = set **of** *этаж*;

Постоянное значение множественного типа строится с помощью конструктора множества, который имеет в точности такую же форму, как и конструкторы, описанные ранее для регулярных и комбинированных типов. Например, при наличии описания

var *сроч_вызовы*: *множ_этаж*;

оператор присваивания

сроч_вызовы :=*множ_этаж* (*подвал, этаж*1*, этаж*3)

зашлет в переменную *сроч_вызовы* значение множественного типа, содержащее три элемента: *подвал, этаж*1*, этаж*3. Оператор присваивания

сроч_вызовы :=*множ_этаж*( )

зашлет в *сроч_вызовы* пустое множество. Нотация конструктора множеств может быть также использована для обозначения преобразования объекта базового типа для множественного типа в объект соответствующего множественного типа. Например, при описании

var *этот_этаж*: *этаж*;

оператор присваивания

сро_вызовы := *множ_этаж*(*этот_этаж*)

зашлет в *сроч_вызовы* множество из одного значения, представленное значением *этот-этаж.*

Над множествами можно производить операции объединения (+), пересечения (*) и разности множеств (—). Например,

сроч_вызовы := *сроч_вызовы* + *множ_этаж*(*этот_этаж*)

добавляет значение *этот_этаж* к множеству *сроч_вызовы,* а

сроч_вызовы := *сроч_вызовы* — *множ_этаж*(*этот_этаж*)

удаляет значение *этот_этаж* из множества *сроч_вызовы.* Отношения над множествами могут проверяться с помощью операций эквивалентности (=), неэквивалентности (< >) и включения множества (<=). Например,

множ _этаж(*под вал, этаж*1) < = *сроч_вызовы*

истинно, если множество *сроч_вызовы* содержит значения *подвал* и *этаж*1. Проверка принадлежности некоторого значения базового множественного типа некоторому множеству осуществляется с помощью оператора **in.** Например,

этот_этаж **in** *сроч_вызовы*

истинен, если множество *сроч_вызовы* содержит значение *этот_этаж.*

Кроме той роли, которую играют множества в приложениях программирования, где производится манипуляция с множествами значений, множественная нотация удобна также и в других областях. Так, например, предположим, что нужно проверить, содержит ли символьная переменная *c* один из символов ‘*A’*, ‘*F’*, ‘*H’*, ‘*Q’* или ‘*Z’*. Традиционно такая проверка выполняется с помощью условия

(*c* =’*A’*) **or** (*c* =’*F’*) or (*c* = ‘*H’*) **or** (*c* = ‘*Q’*) or (*c* =’*Z’*)

что можно сравнить с выполнением этой проверки при использовании множеств:

c in *сим_множ*(‘*A’*, ‘*F’*, ‘*H’*, ‘*Q’*, ‘*Z’*)

Вторая версия, будучи и более удобочитаемой, почти наверняка приведет к более компактному объектному коду.

3.4.2. Реализация множеств

Структура множеств данных обеспечивает механизм описания данных, который обычно представляется битовыми строками. Поэтому множества реализуются как строки из *n* битов, где *n* равно максимальному числу элементов, которое может содержать множество, т. е. *n* является мощностью базового типа для множественного типа. Каждый бит соответствует одному значению базового типа. Его значение равно единице, если это значение является элементом множества, и равняется нулю в противном случае. Однако в отличие от своего битового эквивалента множественные типы очень надежны, так как ответственность за помещение битов в памяти и их проверку берет на себя компилятор, а строгая типизация гарантирует, что в множество будут заноситься значения только из специфицированного базового типа.

Когда базовый тип достаточно мал и допускает представление множества в одном слове вычислительной машины, то множественные операции очень эффективны. Большинство операций можно реализовать с помощью одной или двух машинных команд. Например, объединение множеств — это просто операция «логическое или», а пересечение множеств — операция «логическое и».

3.5. СТРУКТУРЫ ДИНАМИЧЕСКИХ ДАННЫХ

3.5.1. Статические и динамические переменные

Описанные выше регулярные, комбинированные и множественные типы позволяют специфицировать структуры данных, размер и форма которых предопределены. При наличии структуры данных типа *T* экземпляр этой структуры создается с помощью описания переменной вида

var *x*: *T*;

Такую переменную, как *x,* называют статическим объектом данных, потому что она начинает существовать, когда вводится программа, в которой она описана, и существует только во время жизни этой программной единицы. Так как этот объект статический, на него можно ссылаться с помощью приписанного ему имени.

В программировании очень часто требуются структуры данных, размер и форма которых изменяются во время исполнения программы. Их можно построить, объединяя вместе динамические объекты данных. Динамический объект данных создается не с помощью описания переменной, а с помощью обращения к процедуре распределения памяти. По самой своей природе динамические объекты не позволяют ссылаться на них непосредственно по имени, поэтому вместо него должен использоваться механизм косвенных ссылок, предполагающий наличие указателей.

Основные понятия динамических структур данных лучше всего показать на примере. Предположим, что в системе реального времени генерируются аварийные сообщения, которые должны храниться в списке в порядке приоритета, где

```
type аварийное_сообщение = record
                               сообщение: текст;
                               приоритет :аварийный_приор-
                               итет
                              end;
```

Такой список можно построить из динамических записей, определенных следующим образом:

```
type элем_указатель = pointer to список_элем;
     список_элем = record
                      аварийный: аварийное_сообщение:
                      следующий: элем_указатель
                     end;
```

Тип *элем_указатель* есть указательный тип. Объекты этого типа можно использовать для косвенных ссылок на динамиче-

ские объекты, создаваемые с помощью типа *список_элем.* Динамический объект создается при обращении к процедуре распределения памяти с именем, скажем, *новый.* Например, при описании

var *начало, p* : *элем_указатель*;

обращение

новый(*p*)

произведет генерацию нового объекта типа *список_элем* и инициализирует значение *p,* которое будет теперь на него указывать (т. е. адрес заново распределенного динамического объекта приписывается *p*). Указатели и объекты, на которые они указывают, можно отличить друг от друга следующим образом:

p ссылается на сам указатель
p↑ ссылается на обьект, указываемый *p*

Операцию ↑ можно считать операцией, обратной к операции получения ссылки. Следовательно, объект, создаваемый нашим обращением к процедуре *новый,* можно инициализировать следующим образом:

p↑.*аварийный* := *новый_аварийный*;
p↑. *следующий* := *nil*

где *nil* обозначает отсутствие какого бы то ни было объекта и является элементом множества значений любых указательных типов.

Описанная схема манипулирования с динамическими объектами позволяет производить построение произвольных структур во время работы программы, используя указатели для связывания друг с другом динамических записей. В рассмотренном случае нам нужно было построить списковую структуру. Для этого достаточно установить значение компоненты *следующий* так, чтобы оно указывало на следующую запись в списке. На первый элемент в списке указывает статическая переменная указательного типа *начало,* а последний элемент в списке определяется по значению компоненты *следующий,* которая в этом случае равна *nil.*

Для иллюстрации вида обработки при манипулировании с динамической структурой данных рассмотрим операции, которые нужны, чтобы добавить новое аварийное сообщение к описанному выше списку сообщений. Для определенности будем предполагать, что в списке уже содержатся по крайней мере два аварийных сообщения, и приоритет нового списка таков, что новое сообщение должно быть внесено куда-то в середину списка. Внесение нового сообщения, таким образом, потребует

просмотра элементов списка, пока не найдется сообщение с более низким приоритетом, чем новое сообщение, и присоединения нового сообщения в этой точке. Выполняющие эти операции команды можно выразить следующим образом:

```
var новый _аварийный : аварийное сообщение;
     этот, след : элем_указатель;
begin
   этот := начало; след :=этот↑.след; {поиск}
   while след↑.аварийный.приоритет > = новый_аварий-
                                         ный.приоритет
   do begin
      этот := след; след := этот↑.след
   end;
    новый(этот↑.след); этот :=этот↑.след; {вставка}
    этот↑. аварийный := новый_аварийный; этот↑.след :=
    след
end
```

Приведенный пример иллюстрирует главные особенности применения динамических данных. Использованные нотация и семантика взяты из языка Паскаль. В следующих разделах определение механизма динамических данных будет обсуждено более подробно. Но сначала рассмотрим преимущества, которые мы получаем от включения механизма динамических данных в язык реального времени.

Механизмы динамических данных позволяют строить структуры, размер и форму которых можно легко изменять во время работы программы. Поэтому в приведенном выше примере список аварийных сообщений может быть любого размера в рамках имеющейся памяти. Форма списка изменяется каждый раз при добавлении нового элемента, но это может быть сделано очень эффективно, так как требуется только перемещение указателей, а копирования информации в списке не требуется. Чтобы подчеркнуть эти преимущества, рассмотрим, как можно было бы реализовать аварийное сообщение, используя только статические структуры данных.

Простейший подход состоит в хранении сообщений в массиве в порядке убывания приоритета, например,

```
type индекс = целый range 1.. 100;
       аварийный_список = array[индекс] of
                                    аварийное_сообщение;
var список : аварийный_список;
      последний_использованный : индекс;
```

Таким образом, первое сообщение — в элементе список [1], второе — в элементе список [2] и т. д., а последний элемент

списка — это *последний_использованный.* Программа добавления нового элемента в список может иметь следующий вид:

```
var i, этот : индекс;
begin
  этот := 1;
    while список[этот].приоритет> = новый_аварийный.при-
                                                  оритет do
        этот :=этот + 1;
for i :=последний_использованный downto этот do
   список[i + 1] := список[i];
   список[этот] := новый аварийный; оследний_исполь-
                  зованный :=последний_использованный + 1
end
```

Как только в списке определено место, куда должно быть помещено новое аварийное сообщение, оно освобождается сдвигом всех последующих элементов списка на одну позицию вниз. Последнее перемещение связано с очень дорогими операциями копирования, что делает рассмотренное решение практически неприемлемым в системах с сильно ограниченным временем ответа.

Если мы хотим избежать копирования, то порядок элементов в массиве должен обозначаться указателями, в качестве которых могут использоваться индексы. Например,

```
type индекс = целый range 1.. 100;
      список _элем = record
                       аварийный: аварийное_сообщение;
                       следующий: индекс
                         end;
      аварийный_список = array[индекс] of список_элем;
```

Это представление аварийного списка в точности соответствует представлению, использующему динамическую память, за исключением того, что элементы списка располагаются в статическом массиве, а указатели представлены индексными значениями.

Эквивалентный алгоритм для внесения *новый_аварийный* в список имеет вид:

```
var список : аварийный_список;
     начало, этот, след :  индекс;
     новый_аварийный : аварийное_сообщение;
begin
   этот := начало; след := список[этот].след;
   while список[след].аварийный.приоритет > = новый_ава-
                                       рийный_приоритет do
```

```
begin
  этот := след; след := список[этот].след
end;
    получить_место_в_списке(список[этот].след);
    этот := список[этот].след;
     список[этот].аварийный := новый_аварийный;
    список[этот].след := след
end
```

где *получить_место_в_списке* есть процедура, которая выдает индекс свободного места в массиве *список.*

Сравнение кратко очерченных решений задачи об аварийном списке ясно показывает преимущества динамических структур данных. При использовании статической памяти максимальное число аварийных сообщений, которые можно хранить в памяти, должно быть определено заранее и не может быть превышено. Более того, если фактически хранится меньшее число сообщений, то часть памяти пропадает. Использование индексов для связывания элементов в списке требует, чтобы каждый раз при ссылке на элемент списка во время работы программы производилось вычисление адреса. Этого не нужно делать в динамическом случае, где указатели обозначают фактические адреса указываемых объектов. Таким образом, динамическое представление более гибко и более эффективно, чем статическое представление.

Из приведенного обсуждения ясно, что при включении в язык реального времени возможности динамического распределения памяти можно получить значительные выгоды. Однако при точной формулировке такой возможности возникает ряд проблем. К их обсуждению мы сейчас и переходим.

3.5.2. Вопросы проектирования

Включение в язык возможности динамических данных предлагает разработчикам то, что традиционно считается самыми трудными проблемами в проектировании языка. Эти проблемы связаны с тремя потенциальными областями ненадежности. Во-первых, все ссылки на динамические объекты реализуются не непосредственно, а с помощью указателей. Чтобы обеспечить доступ к структуре данных, построенной из таких объектов, должен иметься по крайней мере один ссылающийся на нее статический указатель. Если значение этого указателя будет затерто при ссылке на некоторый другой объект, то первоначальная структура становится недоступной на все время работы. Во-вторых, имя указательной переменной в принципе можно рассматривать либо как сам указатель, либо как объект, на ко-

торый он ссылается. Поэтому при разработке языка нужно ввести четкие обозначения, позволяющие различать оба этих случая; в противном случае может произойти путаница. Это называется проблемой расшифровки ссылок В-третьих, так же, как необходимо предоставить механизм создания динамических объектов, необходимо предоставить и механизм их уничтожения. Но при уничтожении объекта каждый из указателей, который прежде ссылался на объект, останется в рассогласованном состоянии, и последующее его использование может привести к нелепым обращениям к памяти. Это называется проблемой висячих указателей.

Для всех этих проблемных областей существуют решения, но, к сожалению, большинство из них связано с большими издержками времени работы программы, которые совершенно не приемлемы в языках реального времени. Поэтому неизбежен определенный компромисс между надежностью и эффективностью. Впрочем, как будет показано далее, возможна эффективная формулировка, с помощью которой адекватно решаются основные задачи потенциальной ненадежности.

3.5.2.1. Типизация динамических данных

Ясно, что безусловным требованием является применение общих принципов сильной типизации к динамическим данным, так же как и к статическим данным. Из этого прежде всего следует, что определение указательной переменной должно специфицировать тип объектов, на которые она может ссылаться. Поэтому в примере из разд. 3.5.1 указательная переменная *элем_указатель* была описана так:

type *элем_указатель* = **pointer to** *список_элем*;

Таким образом, тип объекта данных, на который ссылаются, всегда может быть определен статически компилятором для любой указательной переменной. Отсюда, имея

var *p,q* : *элем_указатель*;

во время компиляции можно проверить соответствие типов как у прямого присваивания *p*:=*q*, так и у косвенного присваивания p↑:= q↑.

Ранее в этой главе были приведены требования для подтипов массивов и записей. В случае массивов ограничение состоит в том, что указатели на регулярные подтипы должны указывать на объекты только этого подтипа. Например, в случае

type *элем_век* = **array**[*целый*] **of** *элем*;
subtype *век*1 = *элем_век*[0 .. 10];
*век*2 = *элем_век*[5 .. 15];

type *pv*1 **= pointer to** *век*1;
*pv*2 **= pointer to** *век*2;
var *p*1 : *pv*1; *p*2: *pv*2;

присваивание

*p*1:=*p*2

не разрешено, так как типы *pi* и *p2* различны, но присваивание

*p*1↑:= *p*2↑
законно, потому что объекты, на которые указывают *p*1 и *p*2, совместимых подтипов. Следует заметить, что описание указателя на подтип позволяет компилятору выполнить статические проверки типа, так как все указанные объекты имеют одинаковые индексные ограничения, и, таким образом, в этом случае не нужно обращаться к дорогостоящим проверкам во время работы программы.

В случае записей применимы аналогичные соображения. Например, при описании

type *vptr* = **pointer to** *транс*;
cptr = **pointer to** *вид_авт,*

где тип *транс* и подтип *вид_авт* определены так же, как и в разд. 3.3.2, объекты, описанные в

var *vp* : *vptr*;
cp: *cptr*;

различных типов, и поэтому присваивание

vp := *cp*

незаконно. Впрочем, очевидно, что объекты, на которые указывает *cp,* являются подмножеством объектов, на которые указывает *vp,* поэтому это присваивание можно реализовать с помощью явного преобразования типа

vp := *vptr*(*cp*)

Аналогично

cp:=*cptr*(*vp*)

допустимо, но в этом случае преобразование типа связано с дополнительными издержками, так как объект, на который указывает *vp,* должен быть проверен, чтобы удостовериться, что выбран вариант автомобиль. Обратите внимание на то, что преобразование типа разрешено в данном случае и не разрешено

в предыдущем с регулярными указателями, так как значения комбинированных объектов, на которые указывает *ср,* являются подмножеством объектов, на которые указывает *vp,* тогда как в случае массива значения двух регулярных подтипов *век*1 и *век*2 не совпадают.

И наконец, в связи с типизацией динамических данных нужно обсудить вопрос использования рекурсивных описаний. Большинство современных языков программирования с блочной структурой требуют, чтобы все идентификаторы были определены в тексте программы прежде, чем они будут использоваться. Это правило упрощает как разработку компилятора, так и семантику языка. Однако оно представляет большую проблему в связи с динамическими структурами данных, определения которых, как правило, рекурсивны. Например, список аварийных сообщений, рассмотренный в разд. 3.5.1, описывался следующим образом:

```
type элем_указатель = pointer to список_элем; список_элем
     = record
                      аварийный: аварийное_сообщение;
                      следующий: элем_указатель
                    end;
```

где идентификатор типа *список_элем* используется прежде, чем появляется его описание. Одно из решений этой проблемы состоит просто в смягчении строгости описания при использовании правила для указательных типов, но это в дальнейшем приводит к трудностям. Предположим, что идентификатор *список_элем* ранее использовался как имя другого типа, описанного в блоке, включающем блок, в котором появляется наше описание. В этом случае рассматриваемое описание двусмысленно, так как указательный тип может ссылаться на любое из определений *список_элем.* Эту двусмысленность можно обойти только на пути расширения синтаксиса описаний типа, разрешая определение неполных типов, например

```
type элем_указатель = pointer;
список_элем = record
                 аварийный :  аварийное -сообщение;
                 следующий : элем_указатель
               end;
элем_указатель = pointer to список_элем;
```

где первое описание *элем_указатель* просто описывает его как указатель, не специфицируя тип объектов, на которые он указывает.

3.5.2.2. Расшифровка ссылок

До сих пор все примеры программ, в которых использовались указатели, следовали соглашениям языка Паскаль, где все расшифровки ссылок очевидны, т. е. в случае

var *p* : *элем_указатель*;

p обозначает сам указатель, а *p*↑ обозначает объект, на который он указывает. Другими словами, символ ↑ является по сути дела обозначением операции расшифровки ссылки.

У явной расшифровки есть то преимущество, что всегда очевиден точный смысл каждого объекта, на который делается ссылка, и это хорошо согласуется с общим смыслом сильной типизации. Однако, можно возразить, что появление расшифровочной операции в некоторых контекстах отвлекает внимание от основного смысла программы, так как расшифровка является деятельностью низкого уровня, относящейся скорее к аспектам реализации, чем к разработке программы. Предположим, например, что мы имеем следующие описания:

var *s_список* : *список_элем*;
d_список: *элем_указатель*;

В этом случае компонента *аварийнов_сообщение s_списка* обозначается как

s_список. аварийное

но та же самая компонента объекта, на который указывает *d_список,* обозначается как

d_список↑*.аварийное*

Во втором случае явное включение символа f не необходимо, так как может быть только одна интерпретация выражения

d_список. аварийное

Может быть, некоторые возразят, что опущение расшифровочной операции в аналогичных ситуациях ведет к более ясной программе, так как символы ↑ создают ненужные помехи. Подобная аргументация применима к широкому спектру ситуаций, где в случае необходимости расшифровка может быть выведена из контекста. Например, если идентификатор *iptr* был указателем на объекты типа *целый,* то оператор

lptr↑ := 3 может вполне быть записан в виде
iptr : = 3 где расшифровка ясна из контекста.

Ответ на подобные вопросы во многом является делом вкуса. Подводя итог, можно сказать, что явная расшифровка во всех контекстах является, по-видимому, лучшим решением, так как при этом программисту не нужно изучать правила, относящиеся к контекстам, в которых имеет место автоматическая или подразумеваемая расшифровка.

3.5.2.3. Создание и уничтожение динамических объектов данных

Механизмы создания динамических объектов данных относительно просты. По существу, они состоят из обращений к управляющей прохождением программы системе с требованием выделить достаточную для размещения нового объекта память» Это можно сделать либо с помощью нотации обращения к процедуре, например

новый(*vp*)

где *vp* указатель на объекты типа *транс* (см. разд. 3.5.2.1), либо с помощью специальной языковой конструкции, например

vp:= **new** *vptr*

У второго подхода есть то преимущество, что при этом синтаксис может быть расширен так, чтобы допускать (или даже требовать) инициализацию вновь созданного объекта, предоставляя подходящий конструктор; например,

vp := **new** *vptr*(*форд,* 2000, *авт, голубой*)

разместит новую запись типа *транс,* выполнит инициализацию ее компонент и инициализацию указательной переменной *vp,* которая будет теперь содержать ссылку на эту запись.

Уничтожение динамических объектов данных является более трудным делом. С точки зрения программистов простейшая схема состоит в отсутствии какого-то бы ни было явного механизма уничтожения. Вместо этого управляющая система ведет список всех указательных переменных, которые в каждый момент ссылаются на каждый динамический объект. Когда на объект не ссылается ни один указатель, он становится недоступным и поэтому может быть уничтожен автоматически. Это уничтожение либо может произойти, когда объект впервые становится недоступным, либо откладывается на то время, пока имеется достаточно памяти для размещения вновь создаваемых объектов. Эта форма схемы уничтожения, называемая сборкой мусора, очень надежна и решает как проблему недоступности, так и проблему висячих указателей, упомянутую в начале этого раздела. К сожалению, она практически неприемлема для языков-

реального времени, так как связанные с ней издержки времени исполнения очень велики и не предсказуемы.

Другой крайностью является предоставление очень простого механизма, состоящего в процедуре явного уничтожения. Например, обращение *уничтожить* (*vp*) уничтожит объект, на который в данный момент ссылается указательная переменная *vp.* Такая процедура может дать программисту средство точно управлять существованием динамических объектов данных в его программе и при этом может быть реализована с достаточной эффективностью. Эта схема имеет два источника трудностей. Во-первых, может оказаться очень трудно таким образом уничтожить сложные структуры данных, так как на каждый объект структуры должна быть сделана явная ссылка перед его уничтожением. Во-вторых, что более серьезно, после уничтожения каждого объекта остаются висячими указатель, который на него ссылался, и все другие указатели, которые могли содержать ссылки на него. Поэтому этот механизм менее чем идеален для языков реального времени, где надежность и определенность являются делом первостепенной важности.

На практике возможность выбора и уничтожения отдельно конкретных динамических объектов требуется очень редко или не требуется вообще. Поэтому альтернативная стратегия состоит в уничтожении совокупности динамических данных. При описании указательного типа внутри программной единицы появляется потенциальная возможность создания множества или совокупности динамических объектов. При выходе из этой программной единицы описание указательного типа становится больше не действительным и каждый из фактически созданных объектов становится недоступным, потому что все возможные указатели на него исчезли. Тогда совокупность объектов может быть без ущерба автоматически уничтожена системой. Этот механизм надежен и не более дорог при реализации, чем явная процедура *уничтожить.* На самом деле, как будет показано ниже, он может быть реализован почти без затрат в приложениях, где основным показателем является время.

На первый взгляд рассматриваемый механизм кажется несколько ограниченным. В частности, очевидны два недостатка. Во-первых, нет никакого способа уничтожить динамические объекты внутри заданной программной единицы (или области действия). Во-вторых, при выходе из заданной области действия описаний уничтожение всех объектов, которые больше не доступны, обязательно.

Первая проблема в сущности не является никакой проблемой. Необходимость уничтожения динамического объекта данных связана с освобождением памяти для создаваемых новых объектов, когда типы старых и новых объектов различны. Ког-

да они одинаковы, фактическое уничтожение не обязательно. Довольно просто сделать так, чтобы все объекты заданного типа присоединялись к списку свободных, если они больше не требуются, и затем последующие новые объекты могли бы браться из этого списка свободных, без обращения к управляющей системе. Поэтому в любой заданной области управление динамической памятью может быть при необходимости явно запрограммировано. Когда совокупность объектов должна быть уничтожена фактически, чтобы позволить создавать новую совокупность другого типа, то это может быть сделано при выходе из области прежней совокупности.

Вторая проблема принудительного уничтожения и связанных с ним издержек времени работы программы может быть разрешена следующим образом. Во-первых, большинство систем реального времени работает непрерывно, и поэтому трудно представить ситуацию, где требуется создавать новые объекты и не требуется уничтожать. Во-вторых, уничтожения можно избежать, описав указательный тип для соответствующей совокупности в глобальной (или самой внешней) программной единице. В-третьих, как будет показано ниже, уничтожение всегда может быть выполнено очень дешево.

Метод уничтожения на основе области действия описания указательного типа является, по-видимому, наиболее подходящим для языка реального времени, так как он надежен и потенциально эффективен. Но, безусловно, его реализация не должна запрещать использование динамических структур данных в приложениях, где критичным является время. Поэтому в следующем разделе обсуждается реализация управления динамической памятью, которая соответствует такой схеме.

3.5.3. Реализационные аспекты

Реализация механизма динамического распределения памяти описанного выше типа требует добавления к поддерживающей язык системе сопровождения довольно сложной системы управления памятью. Обычно статические данные располагаются либо в фиксированной области памяти, либо, что более распространено, в стеке. Использование стека позволяет безопасно распределять память между несколькими программными единицами (например, процедурами). При входе в программную единицу описанные в ней статические данные располагаются в стеке, а при выходе из единицы данные удаляются, оставляя освободившуюся память программной единице, которая будет работать вслед за ней. Если язык разрешает вложенное исполнение программных единиц, то области данных для нескольких единиц могут быть расположены последовательно в стеке, при-

чем принцип организации стека «последним пришел — первым ушел» гарантирует точную синхронизацию занесения и удаления статических данных с порядком входов в программные единицы и выходов из них. Таким образом, базирующийся на стеке механизм размещения статических данных позволяет эффективно использовать память и прост для реализации. Более того, если язык разрешает рекурсивное исполнение программных единиц, то та или иная форма стековой схемы распределения памяти, как правило, является необходимой.

Возможность простой реализации статических данных базируется на том, что размещение и удаление следуют строгому правилу упорядочения: последним размещен — первым удален. Из предыдущих рассуждений о механизмах динамических данных становится ясно, что никакого такого правила не существует для размещения и удаления динамических данных, где упорядочение в принципе случайно. Поэтому для поддержания механизмов динамических данных требуется система управления памятью, которая организует область памяти таким образом, чтобы пользователь во время работы программы мог явно запрашивать блоки переменного размера. Такая область памяти в программистском обиходе называется *кучей.*

Вначале куча будет состоять из одного непрерывного блока памяти. При рассмотрении запроса на выделение памяти система имеет в куче блок памяти требуемого размера. Затем она отмечает этот блок как «используемый» и передает его адрес запрашивающей программе. Когда блок впоследствии освобождается, система управления памятью возвращает его в пул имеющейся памяти внутри кучи. Стоит обратить особое внимание на то, что рабочие поля выделяются и освобождаются случайным образом. Поэтому, хотя имеющаяся в куче память вначале непрерывна, довольно скоро она становится раздробленной на участки, связанные произвольным образом. Таким образом, система управления памятью должна вести список всех свободных сегментов памяти в куче, чтобы суметь удовлетворить каждый новый запрос о выделении памяти. При разработке таких систем приходится иметь дело с различными проблемами, главная из которых связана с тем, что свободная память дробится на все более и более мелкие блоки. Если на это не обращать внимания, то мы получим такое состояние, когда в достаточно большой общей свободной памяти наибольший непрерывный блок будет таким маленьким, что не сможет удовлетворить даже самые скромные запросы.

Для реализации систем управления динамической памятью были разработаны различные методы. Хорошее описание основных подходов дано Кнутом (1968). Для многих ситуаций реального времени издержки времени работы программы в таких си-

стемах вполне приемлемы. Однако некоторые применения с жестким временным режимом не могут допустить задержек, связанных с распределением и перераспределением динамических данных, особенно если эти задержки могут меняться в зависимости от состояния кучи.

Имеется несколько решений, позволяющих справиться с ситуациями с жесткими временными требованиями. Разумеется, первое — это просто не пользоваться динамическими данными вообще, хотя, как показано в разд. 3.5.1, использование указателей вместо индексов массивов при построении сложных структур данных ведет к повышению эффективности, так как существенно уменьшается работа, связанная с вычислением адресов. В некоторых проектах языков указателям разрешается ссылаться на статические данные так же, как и на динамические данные. Это обеспечивает гибкость и эффективность адресации с помощью указателей без присущих управлению памятью динамических данных издержек. Впрочем, подобные проекты всегда ненадежны, так как указатель легко может стать висячим в результате выхода из программной единицы, где был описан локальный объект данных, на который была сделана ссылка.

Если уничтожение динамических объектов связано с областью действия (т. е. временем жизни) соответствующих описаний указательного типа, как предлагалось в конце последнего раздела, то имеется значительно более хорошее решение. Следует вспомнить, что описание указательного типа предоставляет потенциальную возможность создания совокупности динамических объектов. В общем случае величина этой совокупности не ограничена, и поэтому для ее поддержания требуется система управления всей памятью. Однако если верхняя граница размера совокупности специфицирована в описании указательного типа, то не требуется обязательное размещение совокупности в куче. Вместо этого требуемая область может быть расположена в стеке в области данных программной единицы, в которой был описан указательный тип. Связанные с этим издержки времени работы программы сравнимы с издержками при использовании статических массивов, но преимущества обозначений и эффективность адресации с помощью указательного типа сохраняются. И на самом деле, этот механизм эквивалентен разрешению указательным типам ссылаться на локальные статические объекты, но гораздо более надежен, так как распределение и перераспределение динамических «локальных» объектов данных связаны с описанием указательного типа. В результате ликвидируется возможность оставить висячий указатель. Разумеется, каждый раз при размещении нового объекта нужно

производить проверку, не превышена ли верхняя граница совокупности, но это не дороже, чем проверка границы массивов.

И наконец, следует заметить, что, кроме отмеченных выше проверок для специального случая совокупностей фиксированного размера, единственной проверкой во время работы программы, необходимой при оперировании с динамическими данными, является проверка по обеспечению сохранности указателя со значением *nil.* На многих машинах это можно сделать без каких-либо затрат, объявив значение *nil* незаконным адресом, что ведет к выполнению машинным оборудованием операции «ловли незаконного адреса» при изменении ссылки в указателе *nil.*

3.6. СТРУКТУРНЫЕ ТИПЫ ДАННЫХ В СУЩЕСТВУЮЩИХ ЯЗЫКАХ

Все языки программирования реального времени имеют ту или иную форму механизма массивов, хотя по сравнению с описанным выше общим механизмом многие языки накладывают на него различные ограничения. Из более старых языков CORAL (Вудвард и др., 1970) допускает только одномерные или двумерные массивы, а RTL/2 (Барнес, 1976 и гл. 11) и JOVIAL (Шоу, 1963) фиксируют нижние границы всех индексов массива, которые равны соответственно 1 и 0. Многие из этих старых языков допускают также только ограниченное число компонентных типов. CORAL, например, разрешает только числовые компонентные типы. Проекты языков, появляющихся в последнее время, обеспечивают более общие механизмы массивов; описанный в данной главе механизм очень похож на механизм массивов языка Ада (см. гл. 13).

В отличие от трактовки массивов гораздо больше разнообразия можно найти в подходе к структурированию компонент различающихся типов. JOVIAL, например, имеет возможность структурирования, называемую таблицей, которая является матрицей значений. Каждая строка идентифицируется индексом, но внутри строки индивидуальные значения, которые могут быть различных типов, имеют компонентные имена. Таким образом, таблица является в самом деле массивом записей. Эти табличные механизмы предназначены не только для обеспечения возможности структурирования, но также и для возможности плотной упаковки данных в памяти. В этой главе предполагалось, что отображение структурированных типов на память целевой машины целиком является задачей компилятора. Однако многие приложения в реальном времени требуют, чтобы про-

граммист имел возможность явно распоряжаться этими отображениями. Это рассматривается в качестве самостоятельной проблемы в гл. 7.

Более поздние языки, такие, как RTL/2 и PEARL (Брандес и др., 1970), обладают структурами вида структур языка Алгол 68, которые похожи на описанные в этой главе записи. Проекты современных языков используют нотацию записей, но предлагают другие решения проблемы выделения подтипов. Например, язык Redl (Нестор, 1978) предлагает тип объединение, в то время как Green (Ихбиа, 1978) и Ада предлагают вариантный механизм.

Как правило, проекты новейших языков реального времени используют такой же подход к структурированию данных, что и описанный в этой главе, хотя в большинстве своем разработчики проектов не реализовали множественные типы непосредственно, отдав предпочтение понятию более низкого уровня *битовый* тип, который концептуально определяется как массив логических значений, отображенный на одно слово вычислительной машины.

В большинстве языков имеется та или иная форма указательного механизма. Он реализуется либо с помощью подхода очень низкого уровня, базирующегося на явных операциях косвенной адресации, использующих целые типы, как, например, в языке CORAL, либо с помощью явных ссылок или механизма указательного типа, как в языке RTL/2. Один из аспектов указателей, который мы не обсудили, состоит в том, что они в принципе допускают нежелательные побочные эффекты при обращениях к процедурам и функциям. Когда статические объекты, на которые ссылаются непосредственно, передаются процедуре, любое изменение их значений может быть явно запрещено при передаче их как параметров со спецификацией значения. Это невозможно для объектов, на которые ссылаются косвенно, что является источником потенциальной ненадежности. Некоторые языки, такие, как MESA (Гешке и др., 1977), LIS (Ихбиа и др., 1974) и Ада, модернизируют эту возможность, разрешая специфицировать косвенно объекты, на которые делаются ссылки, как объекты, которые можно только читать.

В проектах старых языков настоящие динамические механизмы по существу не встречались, так как они традиционно считались слишком дорогими для реализации. Однако, как было показано, они не обязательно должны быть такими, и возможны дешевые механизмы динамического распределения памяти. Механизм динамических данных языка Ада соответствует общим принципам разработки, рассмотренным в последней части этой главы.

Глава 4.
КЛАССИЧЕСКИЕ ПРОГРАММНЫЕ СТРУКТУРЫ

4.1. СТРУКТУРИРОВАНИЕ ПРОГРАММ

В предыдущей главе обсуждались спецификации структур сложных данных. Главным обоснованием выбираемого типа разрабатываемых структур была необходимость спецификации данных на множестве различных уровней, так чтобы данные можно было разрабатывать методом сверху вниз, уточняя их на каждом уровне.

Такие же соображения применимы и к спецификации операторов, обозначающих действия, которые должна выполнять программа. Процесс разработки методом последовательного уточнения (Вирт, 1971b) требует, чтобы программа сначала специфицировалась в терминах небольшого числа действий высокого уровня. Каждое действие высокого уровня уточняется затем специфицированием его в терминах действий несколько более низкого уровня. Процесс уточнения продолжается до тех пор, пока вся программа не становится определенной достаточно подробно, так чтобы ее можно было выразить непосредственно на используемом языке программирования.

Для поддержки этой методологии разработки программ язык должен обеспечить конструкции для представления всех обычно встречающихся типов программных структур. Последовательности операторов, которые описывают единичные действия высокого уровня, должны структурироваться на логически самостоятельные единицы. Эти единицы могут реализовываться как блоки, процедуры или функции в зависимости от той роли, которую исполняет данная единица в программе. Существенное значение имеет физическая реализация в конечной программе каждого действия высокого уровня, введенного в процессе разработки в терминах идентифицируемой единицы. Полученная на стадии разработки иерархическая структура должна быть сохранена в тексте программы, иначе потеряется ее ясность. В данном контексте это особенно касается процедур.

На чуть более низких уровнях должны иметься конструкции для спецификации порядка, в котором следует выполнять операторы программы. Это реализуется операторами управления, которые дают возможность условного выполнения операторов, выбора альтернативных операторов и повторного выполнения операторов.

Совместное использование операторов управления, блоков, процедур и функций позволяет написать программу таким способом, который ясно отображает всю разработку программы методом сверху вниз. В нашей книге эти конструкции называ-

ются классическими программными структурами, так как они являются основными строительными блоками традиционных языков программирования. Как мы увидим позже, логическое развитие описанного процесса разработки сверху вниз опирается на понятие абстрактных типов данных. Это развитие поддерживается в современных языках еще одним видом программной структуры, называемым механизмом абстракции. Чтобы подчеркнуть важность этой структуры, рассмотрим ее отдельно в следующей главе, здесь мы обсудим только классические возможности структурирования программ.

Разработка программных структур ставит задачи, аналогичные тем, с которыми мы встретились ранее при разработке структур данных. Эти структуры должны быть надежны, и их смысл должен быть очевиден и однозначен, так чтобы можно было легко понять общую структуру всей программы. Впрочем, они должны также обеспечивать достаточную гибкость, чтобы программист мог выразить свой алгоритм просто и эффективно. Таким образом, приходится идти на определенные компромиссы для согласования этих конфликтующих требований. В разделах данной главы предлагаются возможные решения основных проблем, возникающих в этой области. Сначала рассматриваются операторы управления, затем вводится понятие структуры блока, что, естественно, приводит к обсуждению понятий процедуры и функции. В заключение рассматриваются операторы описаний, хотя они и не характерны для традиционных языков реального времени. Они важны в разработке современных языков, так как используются при определении новых типов данных; роль, которую они играют в обозначении программных структур, менее существенна. В эту главу они включены потому, что очень похожи на описания функций.

4.2. ОПЕРАТОРЫ УПРАВЛЕНИЯ

4.2.1. Основные формы

В примерах программ, которые появлялись в более ранних главах книги, использовались операторы управления, имеющиеся в языке Паскаль. Как будет ясно позднее, выбор этих операторов нельзя назвать идеальным; однако они могут служить для иллюстрации трех основных форм операторов управления.

Условное выполнение операторов обеспечивается конструкцией вида

if *b* **then** *St* **else** *Sf*

где оператор *St* выполняется, если логическое выражение *b* истинно, а оператор *Sf* выполняется, если *b* ложно. Часть **else** необязательна и может быть опущена.

Выполнение оператора из группы альтернатив обеспечивается оператором **case,** имеющим форму

```
case x of
xa := Sa;
xb := Sb;
   .
   .
   .
end
```

где *x* — переменная выбора, а *xa, xb, ...*— постоянные значения того же типа, что и *x.* Оператор *Sa* выполняется, когда *x = xa,* оператор *Sb* выполняется, когда *x = xb,* и т. д. И наконец, повторное выполнение оператора можно осуществить с помощью оператора **while**, имеющего вид

while *b* **do** 5

где оператор *S* выполняется повторно, пока истинно логическое выражение *b.* Это выражение перевычисляется перед выполнением *S.*

Приведенные три оператора обеспечивают программиста четкими возможностями представления трех основных форм структур управления. На самом деле в Паскале имеются еще два дополнительных вида структур управления для повторения: оператор **repeat** и оператор **for**. Оператор **repeat** очень похож на оператор **while,** и его включение в язык обосновать несколько затруднительно, так как он в общем избыточен. Хотя оператор **for** также избыточен в том смысле, что его можно реализовать с помощью цикла **while,** он является существенно другим видом управляющего цикла, и его включение в язык, по-видимому, необходимо. Конструкции циклов обсуждаются подробно в разд. 4.2.3. И наконец, стоит отметить, что в тех случаях, когда управлению подлежит группа операторов, они должны быть объединены в один составной оператор заключением их в ключевые слова **begin** и **end.** Как будет показано в следующем разделе, это приводит к серьезным трудностям.

4.2.2. Разработка операторов управления

Разработка операторов управления должна удовлетворять трем базовым критериям:

(i) Смысл оператора должен быть очевидным и однозначным.

(ii) Конструкция оператора должна допускать иерархическую структуру вложенных операторов, которая должна быть ясна из текстуального расположения программы,

(iii) Нужно, чтобы конструкцию оператора можно было легко модифицировать.

Прежде чем предложить конкретные конструкции, удовлетворяющие этим критериям, полезно внимательнее рассмотреть оператор **if** из языка Паскаль, так как в нем имеется ряд существенных погрешностей.

В языке Паскаль вложенная конструкция if

if *b*1 **then if** *b*2 **then** *Sa* **else** *Sb*

неоднозначна, так как возможны две различные интерпретации. Оператор *Sb* может выполняться либо когда *b*1 ложно, либо когда *b*1 истинно, но *b*2 ложно. Разумеется, описание языка задает правила для разрешения этой неоднозначности (правильна вторая интерпретация), но нас интересует не это. Тот факт, что форма оператора неоднозначна, является серьезным недостатком с точки зрения надежности, так как даже опытный программист может случайно ошибиться. Например, когда рассматриваемый вложенный оператор переписан в форме

```
if b1 then
   if b2 then Sa
   else
    Sb
```

то данная неоднозначность заметна гораздо меньше. Судя по расположению текста, правильной может показаться интерпретация «если *b*1 истинно, то выполнить оператор «**if** *b*2 **tben** *Sa*», иначе выполнить *Sb*», что на самом деле не верно. Ясно, что оператор **if** языка Паскаль не удовлетворяет первому критерию.

Текстуальное расположение оператора **if** языка Паскаль осложняется в связи с различающимися формами, которые могут встретиться в результате включения составных операторов. Рассмотрим пример:

```
if b1 then
  if b2 then S
```

Используя скобки составных операторов, данный оператор можно с успехом переписать в виде

```
if b1 then  begin
   if b2 then
   begin
     S
   end
end
```

т. е. в форме, отличающейся от предыдущей. Как правило, включение **end** для обозначения конца оператора удобно, так как явно ограничивает область управления, но символы **begin** не добавляют никакой визуальной информации о структуре, а только вносят ненужные помехи. Таким образом, оператор **if** языка Паскаль не полностью удовлетворяет требованиям второго критерия.

Последний критерий разработки относится к модификации программ. Здесь также необходимость использовать составные операторы вносит дополнительные трудности. Предположим, что нам нужно включить еще один оператор в конструкцию

```
if b then
   Sa
```

Просто приписать этот оператор после *Sa* недостаточно, так как конструкция

```
if b then
   Sa; Sb
```

имеет другой смысл. Поэтому наряду с внесением дополнительного оператора нужно изменить и сам оператор **if:**

```
if b then
begin
   Sa; Sb
end
```

И здесь оператор **if** языка Паскаль построен так, что не будучи достаточно внимательным, можно легко совершить ошибку. Все отмеченные выше трудности с оператором if языка Паскаль связаны с тем, что все действия, которыми управляет оператор, должны быть обозначены одной-единственной операторной конструкцией. Более удобно такое построение операторов управления, при котором разрешается включать в область управления не один оператор, а их последовательность, Из этого следует, что у каждого оператора управления должны иметься свои явные заключающие скобки. Например, простой оператор **if** желательно записать в виде

```
if b then
   S
end
```

где *S* обозначает один или более операторов. Явные заключающие скобки однозначно указывают область действия оператора и также способствуют простому унифицированному расположению текста. Кроме того, пропадает возможность какой бы то ни было двусмысленности.

Небольшое затруднение при этом подходе к разработке связано с выбором заключающих скобок для каждого оператора управления. Анонимное ключевое слово end, которое мы употребили в предыдущем примере, не может считаться лучшим решением. В идеальном случае каждая заключающая скобка должна быть уникальной для каждого типа оператора управления. Это увеличивает ясность программы и позволяет компилятору лучше провести диагностику в случае недостающей скобки. Впрочем, введение набора новых ключевых слов нежелательно, так как оно увеличивает сложность компилятора и число ключевых слов, которые должен выучить программист. Последнюю трудность можно обойти с помощью введения стандартного правила именования каждой заключающей скобки, например, используя обратное написание открывающего ключевого слова, т. е. **if** ... **fi**, **case** ... **esac**. Другое решение не требует введения новых ключевых слов; используется последовательность существующих ключевых слов, например **if** ... **end if**, **case** ... **end case**. Хотя последнее решение несколько многословно, его, вероятно, можно назвать лучшим, так как оно минимизирует число ключевых слов в языке и не заставляет использовать трудночитаемые ключевые слова, такие, как **esac**, **od** и т. д.

4.2.3. Примеры конструкций

В данном разделе будут даны примеры конкретных конструкций для каждой из базовых структур управления. Они приводятся только в самой общей форме; почти весь «синтаксический сироп», который обычно сопровождает практические языки, опущен. Примеры использования этих конструкций можно найти в оставшихся главах этой части книги.

Во-первых, условное выполнение оператора может быть представлено оператором **if** в следующей форме:

```
if b then
  S1
else
  S2
end if
```

где *Si* обозначает произвольную последовательность операторов. Возможная трудность связана здесь с вложенными операторами **if**, так как расположение текста становится излишне громоздким. Например,

```
if b1 then
S1
```

```
else
   if b2 then
     S2
   end if
end if
```

Здесь может показаться, что оператор *S*2 находится на более глубоком уровне вложенности, чем оператор *S*1, но логически это операторы одного уровня. Данную проблему можно решить введением части **elsif.** Например, последнюю запись можно переписать в виде

```
if b1 then
   S1
elsif b2 then
   S2
end if
```

В общем случае может быть включено произвольное число частей **elsif,** что дает оператору **if** специфическую «гребенчатую» структуру, т. е.

```
if b1 then
   S1
elsif b2 then
   S2
elsif b3 then
   S3
   .
   .
   .
else
   Sn
end if
```

Эта форма оператора **if** позволяет работать с четким и согласованным расположением и исключает двусмысленности.

Альтернативный выбор может быть представлен оператором **case** в форме, близкой к версии языка Паскаль, с требованием дополнительного ограничителя, отмечающего конец последовательности каждого оператора **case.** Это можно сделать с помощью добавочных круглых скобок, что видно в следующем примере. Такая форма оператора выбора хорошо дополняет выборную часть вариантной записи, рассмотренной в гл. 3.

```
case x of
   x1 : (S1);
   x2: (S2);
    .
    .
    .
   xn: (Sn);
end case
```

Здесь x — переменная выбора, xi — метки выбора, Si — произвольные последовательности операторов.

Повторное выполнение оператора можно представить, используя универсальный общий синтаксис для обозначения цикла необязательным префиксом, указывающим управление циклом **while** или **for,** т. е.

```
[префикс цикла]
do
  S1
end do
```

Если не задается никакого префикса, то повторное выполнение последовательности операторов происходит все время. Эта форма часто требуется при спецификации циклических процессов. Условное выполнение цикла можно получить, присоединяя префикс **while,** т. е.

```
while b do
  S1
end do
```

где b — логическое выражение. Индексное управление циклом получается при присоединении префикса for, т. е.

```
for i in x do
  S1
end do
```

Эта форма индексной спецификации в операторе **for** несколько необычна. Она введена с двойной целью. Если x является дискретной областью, как, например, в случае

```
for i in 1..10
```

то i при каждом повторении цикла принимает последовательные значения 1, 2, 3 ... 10. Если же x есть множественная переменная, то i при каждом повторении цикла принимает значение каждого очередного элемента этого множества. Порядок выбора каждого элемента из множества не существен. Заметим, что в первом случае размер шага всегда считается равным 1. Поэтому тип индекса может быть любым дискретным типом, включая и перечислимый.

4.2.4. Ненормальное завершение операторов повторения

Кратко описанные в предыдущем разделе формы циклических операторов управления являются чистыми формами циклических конструкций. Их отличительная особенность состоит в том, что у них одна точка входа и одна точка выхода. Эта особенность считается необходимой для всех конструкций, используемых в структурном программировании, так как она позволяет выполнять иерархическую декомпозицию сложных программных структур и упрощает методы доказательства (по этому вопросу см., например, классическую работу Дейкстры, 1972). Впрочем, в контексте языков программирования реального времени могут потребоваться более гибкие средства завершения цикла, особенно в тех случаях, где требуется очень эффективный объектный код.

Покажем это на простом примере. Предположим, что необходимо произвести поиск в массиве *a*, состоящем из *n* объектов типа *элем,* чтобы найти некоторый специфицированный элемент. Обычный способ просмотра массива фиксированного размера состоит в использовании цикла **for.** Но в нашем случае просмотр массива нужно закончить сразу же после обнаружения искомого элемента. «Чистое» решение этой задачи поэтому состоит в том, чтобы воспользоваться циклом **while** следующим образом:

```
i := 1; найден := ложь;
while(i <=n)and not найден do
    найден := a[i] = треб_элем;
    i := i + 1
end do
```

Цикл **while** пригоден потому, что в одном-единственном логическом выражении можно объединить любое число условий завершения, и поэтому выполнение цикла может быть прекращено либо при обнаружении требуемого элемента, либо при достижении конца массива. Недостатком этого подхода является необходимость описания индексной переменной обычным способом и хранения ее в памяти. Внутри цикла ее приращение должно определяться с помощью явного оператора присваивания. При использовании же цикла **for** компилятору легче генерировать гораздо более эффективный объектный код. Это связано с тем, что индексная переменная в цикле **for** обычно считается локализованной в цикле. Поэтому компилятор может хранить эту переменную в регистре, эффективно осуществлять ее приращение и даже оптимизировать обращения к массиву.

Гибкость циклических операторов управления может быть значительно увеличена за счет включения в язык оператора **exit.** В результате выполнения оператора **exit** содержащий его

цикл заканчивает свою работу сразу же. Поэтому, используя этот оператор, последнюю программу можно закодировать более эффективно:

```
for i in 1..n do
  найден := a[i] = треб_элем;
  exit when найден
end do
```

Второе решение не безупречно в том смысле, что имеются две точки выхода из оператора **for**. А в остальном оно не менее ясно, чем предыдущая версия с циклом **while**.

Неумеренное использование оператора **exit** может привести к созданию неясных программ, особенно когда в одном и том же цикле имеется несколько таких операторов. Заметим, однако, что, хотя введение такого оператора и противоречит правилам чистого структурного программирования, его введение гораздо менее опасно, чем использование для выхода из цикла обычных операторов типа **goto**. На самом деле в некоторых случаях версии программ с использованием операторов **exit** могут оказаться более ясными и более легкими для понимания, чем соответствующие чистые версии (см., например, Цан, 1974). Поэтому использование оператора **exit** делает язык более гибким и эффективным. В связи с этим можно сделать вывод, что эта особенность в языках реального времени желательна.

4.3. БЛОЧНАЯ СТРУКТУРА

4.3.1. Предпосылки

Конструкция программы в своей наиболее общей форме логически может быть разделена на две части: спецификация данных программы и спецификация действий программы. В системе обозначений языка Паскаль, таким образом, все программы имеют следующую общую структуру:

```
program имя;
  конст...
  тип….       данные
  перем...
begin
    …
    …         действия
    …
end
```

где спецификация данных состоит из описаний констант, описаний типа и описаний переменных, а спецификация действия состоит из последовательности выполнимых операторов.

Большие программы эта простая структура отражает неадекватно в нескольких отношениях. Во-первых, в ней никак не отмечены отдельные логические единицы, введенные в процессе ее разработки. Поэтому для программ, которые, возможно, были разработаны так, что обладали модульной структурой, эта первоначальная структура теряется в конечном монолитном программном тексте, что делает ее трудной для понимания и сопровождения. Во-вторых, необходимость иметь постоянное место в памяти для всех переменных, используемых в программе, вызывает расточительный расход памяти. Как правило, не все объекты данных должны быть активными в каждый момент времени жизни программы; поэтому некоторые переменные, возможно, могли бы быть размещены в одном и том же месте памяти. В-третьих, так как все объекты данных имеют одну и ту же именную область действия, т е. они известны во всей программе, им всем должны быть даны различные имена. Из этого следует, что компилятор должен построить очень большую лексическую таблицу, что приводит к снижению скорости компиляции, а также к чрезмерному расходу памяти. Кроме того, программист может оказаться вынужденным выбирать неестественные идентификаторы, затемняя этим смысл программы. В-четвертых, описания объектов данных текстуально удалены от операторов, которые их обрабатывают, что затрудняет чтение программного текста. И наконец, в тех программах, где сложное действие нужно выполнять в нескольких различных точках, выполняющая это действие последовательность операторов должна выписываться заново в каждой точке. При этом напрасно тратятся как место в памяти, так и усилия программиста; повышению ясности программы это также не способствует.

В классических проектах языков эти проблемы решаются путем введения понятий подпрограммы и блочной структуры. Подпрограммная единица может выступать в одной из следующих четырех форм:

(i) простой блок
(ii) процедура
(iii) функция
(iv) оператор

Обычно все подпрограммные единицы могут использоваться в блочной структуре, хотя в конкретных языковых проектах их использование может ограничиваться из соображений эффективной реализации. Подробнее это обсуждается в дальнейших

разделах. В данном разделе мы разбираем понятие блочной структуры на примере простых блоков. Далее в разделах последовательно рассматриваются более важные подпрограммные единицы: процедуры, функции и операторы.

4.3.2. Блоки

Как и главная программа, блок состоит из двух частей: спецификации всех требующихся внутри блока локальных данных и последовательности операторов, обозначающих действия, которые должен выполнить блок. Сам по себе блок может быть описан в любом месте программы, где может быть написан оператор. При входе в блок создаются все описанные внутри блока объекты данных, и затем выполняется последовательность операторов.

Блок обеспечивает две основные возможности. Во-первых, он дает средство для создания локальной памяти. В качестве простого примера предположим, что каждой из двух переменных нужно присвоить значение другой переменной. Чтобы это сделать, необходимо ввести в качестве промежуточной памяти третью временную переменную. Реализующий эту функцию блок может быть описан следующим образом (здесь ключевое слово **block** обозначает начало блока):

```
block
  var врем : тип_элем;
  begin
    врем := элем1
    элем1 :=элем2;
    элем2 := врем
  end;
  …
```

При выходе из блока выделенная для переменной *врем* память освобождается и может использоваться для других нужд. Заметим, что переменная *врем* строго локализована в блоке и не доступна извне.

Вторая возможность, которую обеспечивает блок, состоит в введении новой именной области действия. Все описанные в объемлющей главной программе имена внутри блока видимы (т. е. на них можно ссылаться и их можно использовать). Однако, если имя описано внутри блока локально, то оно отменяет любое описание в главной программе, которое определяет то же самое имя. Это является чрезвычайно полезной возможно-

стью, так как позволяет программисту использовать часто встречающиеся имена (например, *i, j, k* и т. д.) внутри блока, не боясь нечаянно изменить значение переменной, которая где-то в другом месте программы имеет то же самое имя. Поясним это на примере следующей небольшой программы:

```
program имена;
   var i, j : целый;
begin
   i:=0; j:=0;
   block
      var i: целый;
   begin
      i:=1; j:=1;
   end;
   {i = 0, j=1, здесь}
 end
```

Здесь две переменные *i* и *j* описываются в главной программе, и им присваиваются нулевые значения. Далее описывается блок с локальной переменной, имя которой также *i.* Внутри этого блока все ссылки на имя *i* относятся к этой локальной переменной, так как описание i в главной программе отменяется, однако ссылки на *j* относятся к переменной *j*, которая описана в главной программе. Поэтому при выходе из блока переменная *i* главной программы остается неизменной, а переменная *j* получает значение 1.

Так как блок может быть написан везде, где может быть написан оператор, блоки могут вкладываться друг в друга, образуя иерархическую программную структуру. Это является первым шагом по направлению к реализации логических единиц, вводимых в процесс разработки программы. Возможность образовать структуры с вложенными блоками требует более точного определения области действия имен объектов. Классические правила именной области действия для блочной структуры могут быть сформулированы следующим образом:

(i) Областью действия некоторого имени является блок, в котором это имя описано, и все блоки, содержащиеся в этом блоке, при соблюдении правила (ii).

(ii) Когда имя, описанное в блоке A, заново описано в некотором блоке B, содержащемся в A, то блок B и все блоки, заключенные в B, из области действия описания этого имени в A исключаются.

Покажем работу этих правил области действия на следующем эскизе программы. Внутри каждого блока комментарии указывают, какие имена видимы в операторной части этого блока.

```
program правила_области;
  var x, y : плавающий;
  begin
    {x, y : плавающий - видимы}
    block
      const x = 0;
      var c : сим;
    begin
       {y : плавающий, c: сим;
       константа x - видимы}
       block
          type y = любой_тип;
       begin
         {константа x, c : сим;
          тип y - видимы}
       end;
      {y : плавающий, c : сим
       константа x - видимы}
    end:
    {x, y : плавающий - видимы}
  end
```

Приведенные правила области действия приняты почти во всех классических языках программирования. Они определяют то, что специалисты называют открытой областью действия для блоков. Это означает, что внутри блока все имена, описанные в обрамляющих блоках, автоматически видимы (при условии, что они не отменены новым описанием). Для блоков это представляется разумным подходом. Блоки используются в основном как средство введения временных локальных переменных надежным и эффективным способом, а не как основной инструмент структурирования программ. Однако, когда те же самые правила области действия применяются к другим подпрограммным единицам, их полезность менее очевидна. Это обсуждается далее в разд. 4.4.3.

4.3.3. Реализация блоков

Реализация блоков требует, чтобы память для локальных переменных распределялась при каждом входе в блок и затем освобождалась при выходе из блока. Это, однако, не означает, что во время выполнения программы обязательно должна производиться какая-то лишняя работа, так как компилятор

может определить заранее порядок входов и выходов из блоков. Поэтому адреса всех локализованных в блоках переменных могут быть вычислены во время компиляции в форме сдвига от начала области данных объемлющей главной программы (или процедуры, функции и т. д.).

Реализация правил именной области действий в компиляторе проста. Лексическая таблица организуется компилятором в виде стека таким образом, что при необходимости найти некоторое имя эта таблица просматривается в направлении от последних записей к самым ранним. Поэтому первым всегда находится имя, которое используется в смысле самого последнего описания; предыдущие описания при этом всегда не учитываются. Когда компилятор доходит до начала блока, в лексической таблице делается отметка, так что при достижении конца блока все описанные внутри этого блока имена могут быть вытолкнуты из стека. Это обеспечивает невозможность обращения к локальным именам блока извне блока. Дополнительное преимущество этой схемы состоит в том, что компилятору не нужно хранить каждое описанное в программе имя в лексической таблице все время; требуется хранить только те имена, к которым в данный момент возможны обращения. Поэтому, экономя память во время компиляции, мы получаем возможность компилировать гораздо более объемные программы.

4.4. ПРОЦЕДУРЫ

4.4.1. Понятие процедуры

Описанный в предыдущем разделе механизм блочной организации предлагает только частичное решение проблем, изложенных в разд. 4.3.1. В частности, он предоставляет лишь минимальную возможность для описания программы в терминах композиции логически различных единиц и по-прежнему требует повторного описания сложных действий, когда они должны выполняться в разных точках программы. Естественное обобщение понятия блока состоит в том, чтобы рассматривать блок не как описание непосредственно выполняемых команд, а скорее как описание действия, которое необходимо выполнить. Связав с этим действием имя, можно было бы возбудить требуемое действие простой ссылкой на это имя в тех местах программы, где требуется выполнить само действие. Такое описание называется процедурой, а ее возбуждение — вызовом процедуры.

Процедуры являются важнейшим элементом любого языка программирования. Первоначально они были введены с целью решения проблемы многократного дублирования одинаковых совокупностей команд. Однако в настоящее время им придается

такая же (если не большая) важность в качестве средства структурирования программы. При разработке программы методом последовательного уточнения каждое действие высокого уровня может быть обозначено в тексте программы ясно с помощью оператора вызова процедуры. Далее каждый этап уточнения будет состоять из определения процедур, которые вызывались на предыдущем этапе, возможно, в терминах обращений к процедурам более низкого уровня. Таким образом, программа организуется в виде иерархии определений процедур. Каждая процедура может быть выражена в текстуально компактном виде, что делает всю программу более понятной и удобной для сопровождения.

Описание процедуры записывается с сохранением той же самой основной формы, которую мы указали раньше для блока. В первой строке описания процедуры специфицируются имя процедуры и ее параметры, когда такие имеются. Параметры специфицируют интерфейс ввода-вывода процедуры с внешней средой. Остальная часть процедуры оформляется точно так же, как и блок. Само описание процедуры помещается в описательной части объемлющей программной единицы

В качестве примера использования процедуры рассмотрим следующую программу чтения последовательности чисел, заканчивающейся числом с нулевым значением, которая выдает на печать среднее арифметическое чисел последовательности:

```
program среднее;
   var x, сумма, счетчик : целый;
      последний : логический;
  procedure значение_процесса (in значение : целый;
                                      out последний : логический);
   begin
      последний := значение = 0;
      if not последний then
         сумма := сумма + значение; счетчик := счетчик + 1
      end if
   end;
 begin
   сумма := 0; счетчик : = 0; последний : = ложь;
   while not последний do
      читать(x); значение_процесса(x, последний)
  end do;
   писать(сумма/ счетчик)
end
```

Прежде всего для этой программы нужно объяснить способ спецификации параметров в процедуре *значение_процесса.* Перед параметром *значение* стоит ключевое слово **in**, которое

указывает, что этот параметр используется для ввода значения в процедуру. И наоборот, стоящее перед параметром *последний* ключевое слово **out** указывает, что этот параметр используется для возвращения результата в среду, откуда произошло обращение (вызывающую среду). Хотя это выглядит вполне разумным, в связи с такой спецификацией возникает несколько важных вопросов. Что на самом деле означают ключевые слова **in** и **out** и какие другие еще нужны классы параметров? Выбор механизма передачи параметров является, по существу, первым из двух главных вопросов, которые необходимо рассмотреть при формулировке механизма процедуры. Более детально он обсуждается в разд. 4.4.2.

Второй вопрос, который нужно рассмотреть, касается применения правил именной области действия к описаниям процедуры. Проект приведенного выше примера программы на самом деле далеко не идеален. То, как процедура *значение_процесса* неявно взаимодействует со средой, модифицируя глобальные переменные *сумма* и *счетчик,* вместо того, чтобы явно воздействовать на них через параметры, не является примером хорошей практики программирования. Такое взаимодействие возможно лишь тогда, когда к описаниям процедур применяются классические правила именной области действия для блочной структуры. Могли бы быть приняты менее либеральные правила именной области действия, что не позволило бы процедуре автоматически получить свободный доступ к среде, откуда происходит обращение. Эта проблема рассматривается в разд. 4.4.3.

4.4.2. Параметры

Точная формулировка механизма передачи параметров для процедур при разработке языка является трудным вопросом, для которого не существует общепринятых решений. В данном разделе мы приведем основные результаты, относящиеся к разработке языков реального времени.

Описание каждого параметра в заголовке процедуры должно специфицировать имя, по которому на параметр будут производиться ссылки в теле процедуры, его тип и класс. Например, заголовок процедуры

procedure *пример*(*out фпар* : *любой_тип*);

утверждает, что у процедуры *пример* имеется один параметр с именем *фпар,* тип которого *любой_тип* и класс **out.** Спецификация типа параметра существенна в сильно типизируемом языке, она позволяет применять правила проверки типа к вызовам процедуры. Таким образом, в вызове *пример* (*моя_пер)*

переменная *моя_пер* должна быть эквивалентна по типу с соответствующим формальным параметром *фпар.*

Строгая проверка соответствия типа между формальными и фактическими параметрами является основным требованием в надежном языке. При ее включении в проект языка существенных трудностей не возникает, однако, если стремиться к сохранению гибкости языка, здесь нужно проявлять определенную осторожность. Например, если система типизации такова, что размер массива является частью его типа, то строгое применение правил сильной типизации не позволит писать процедуры, оперирующие с массивами переменной длины (сравните с языком Паскаль). Подобные проблемы можно устранить, если смягчить применение правил сильной типизации к проверке соответствия между формальными и фактическими параметрами. Это, впрочем, нежелательно из тех соображений, что всегда предпочтительно избегать «специальных случаев». Гораздо лучше попытаться построить такую базовую систему типизации, в которой подобные трудности никогда бы не возникали (см. гл. 3).

Стоит подчеркнуть еще раз, что при существующем машинном оборудовании система типизации языка реального времени должна быть статической, так чтобы по возможности все проверки типа могли быть проведены во время компиляции. Некоторая добавочная гибкость обеспечивается введением понятия подтипа, что может повлечь за собой некоторые издержки времени работы программы, однако в общем случае основную работу по проверке типа выполняет компилятор.

В результате применения описанного общего подхода автоматически исключаются из проекта системы параметров различные интересные возможности. В частности, параметризация типов могла бы разрешить построение процедур самого общего назначения, но тогда все проверки типа должны были бы производиться во время работы программы. Следующие поколения вычислительных машин, возможно, сумеют обеспечить аппаратную поддержку для проверки типа, что позволит вывести из употребления представленную здесь статическую систему типизации.

В отличие от типов параметров спецификация классов параметров связана с дилеммой, для которой не существует идеального решения. Концептуально класс параметра определяет, какую из трех возможных функций выполняет параметр:

(i) передача значения из среды обращения в процедуру — ввод;
(ii) возврат значения из процедуры в среду обращения — вывод;
(iii) как (i), так и (и) — ввод-вывод.

Язык должен разрешать программисту специфицировать то, как он намеревается использовать параметр. Это важно как для того, чтобы предотвратить случайное неправильное использование процедуры из вызывающей среды, так и для того, чтобы исключить ошибки в самой процедуре (например, случайное присваивание значения параметру ввода). К тому же, разумеется, явное указание функции параметра в заголовке процедуры помогает понять работу процедуры.

Ясно, что спецификация класса параметра важна. К сожалению, не так-то просто приписывать выбранному множеству классов точную и реализуемую семантику. По существу, возможны два подхода к спецификации классов параметров. Первый состоит в спецификации семантики каждого *класса в* абстрактных, не зависящих от реализации терминах, а второй - в спецификации для каждого класса механизма фактической реализации.

Рассмотрим сначала абстрактный подход. Классы параметров можно определить так, чтобы они соответствовали каждой из перечисленных выше трех функций. Для каждого из этих классов можно дать подходящие имена и определения:

in Внутри процедуры формальный параметр работает как локальная константа, которой присвоено начальное значение, задаваемое соответствующим фактическим параметром в вызове процедуры. Фактическим параметром может быть любое выражение, вырабатывающее значение того же типа, что и у формального параметра.

out Внутри процедуры формальный параметр работает как локальная переменная. Ее значение присваивается соответствующему фактическому параметру в вызове процедуры как результат выполнения процедуры. Фактический параметр должен быть переменной того же типа, что и формальный параметр.

inout Внутри процедуры формальный параметр работает как локальная переменная, которой присваивается начальное значение, задаваемое соответствующим фактическим параметром в вызове процедуры. Конечное значение этой локальной переменной присваивается соответствующему фактическому параметру как результат выполнения процедуры

В большинстве ситуаций эти определения могут адекватно и однозначно специфицировать поведение процедур независимо от фактической реализации механизма передачи параметров. Заметим, однако, что определения для **out** и **inout** на самом деле не устанавливают, когда значение локального формаль-

ного параметра присваивается соответствующему фактическому параметру. Эта неопределенность оставлена нарочно, чтобы иметь возможность применять различные механизмы передачи параметров. Для понимания, почему это необходимо, требуется описать основные механизмы передачи параметров.

По существу, имеются два метода передачи параметров, которые пригодны для языков программирования реального времени копирование и при помощи ссылок. Рассмотрим копирование на примере его наиболее общего класса **inout:** при входе в процедуру задаваемое вызовом значение фактического параметра копируется в локальную переменную. Все ссылки на формальный параметр в процедуре относятся теперь к этой локальной переменной. При выходе значение локальной переменной копируется обратно в фактический параметр. Этот механизм обычно называется механизмом с результатом-значением. В случае же передачи с помощью ссылок при входе в процедуру в локальную переменную копируется адрес фактического параметра. Все ссылки на формальный параметр в процедуре затем переводятся при компиляции в косвенные ссылки на сам фактический параметр с помощью этого адреса.

Необходимо разрешить компилятору использовать любой из этих механизмов для реализации абстрактных классов **in, out** и **inout,** чтобы как очень малые, так и очень большие объекты могли передаваться эффективно. В языках реального времени часто требуется упаковать несколько малых объектов (например, логических) в одном машинном слове. В этом случае передача одного такого малого объекта по ссылке непосредственно не возможна. Поэтому для поддержания эффективности нужно использовать копирование. И наоборот, передача копированием больших объектов, таких, как массивы, без нужды расходует память и время. Поэтому в этом случае следует воспользоваться передачей с помощью ссылок.

В большинстве случаев выбор механизма передачи параметров не влияет на семантику программы. Но, к сожалению, встречаются и особые случаи. Рассмотрим следующую процедуру, использующую нелокальную переменную *y*:

```
var y: целый;
procedure доб_y_1(inout x : целый);
begin
  x:=x+1;
  x:= x + y
end;
```

При выполнении пары операторов

```
y:=1;
доб_y_1(y)
```

окончательное значение *y* будет 3 в случае копирования и 4 в случае передачи по ссылке. Такие ситуации, как эта, называются двойственностью именования, их возникновение связано с наличием двух путей доступа к одной переменной. Передача параметров копированием не может вызвать такой эффект, а передача с помощью ссылок - может.

Рассмотренный эффект обычно считается программистской ошибкой, теоретически он может быть обнаружен компилятором. Поэтому можно было бы утверждать, что двойственность именования не является достаточной причиной для отказа от абстрактного подхода к определению классов параметров. Однако при использовании общих переменных в обрабатывающей несколько процессов системе возникает еще одна и значительно более трудно обнаруживаемая ошибка.

Разделение времени между несколькими задачами подробно обсуждается в гл. 6. Для иллюстрации наших проблем рассмотрим очень простой пример. Предположим, что процедура с именем *подсчет* должна использоваться для обновления совокупности счетчиков, доступных нескольким процессам. Так как процедура может быть вызвана одновременно двумя процессами, команды процедуры написаны в виде специального раздела, охраняемого семафором *s*, т. е.

```
type счетчик = целый range 0..1000;
procedure подсчет(inout c : счетчик);
begin
   надежный(s);
   c := c + 1;
   освободить(s)
end;
```

Введенный семафор гарантирует, что если два процесса одновременно требуют сделать приращение одному и тому же счетчику, то целостность счетчика будет поддерживаться. Это означает, что при наличии описания

```
var c1 : счетчик;
```

(где *c*1 - глобальная переменная, которой присвоено начальное значение 0), если два процесса обращаются к процедуре *подсчет* одновременно, например

```
process один;    process второй;
.                .
.                .
.                .
подсчет(c1)      подсчет(c1)
.                .
.                .
.                .
end              end
```

то независимо от того, когда каждый процесс вызывает процедуру *подсчет,* конечное значение *c*1 будет равно 2. (Проверьте, будет ли?)

На самом деле приведенная программа будет работать правильно лишь тогда, когда общая переменная *c*1 будет передаваться по ссылке. При использовании копирования и одновременном обращении к процедуре *подсчет* конечное значение *c*1 будет равно 1, так как передача параметра происходит не при взаимном исключении процессов, хотя сами обновления друг друга взаимно исключают. Поэтому выбор реализации передачи параметров в мультипрограммных системах может влиять на выполнение программы самым неожиданным образом. Конечно, рассмотренный пример несколько надуман, и в этом простом случае проблема может быть легко решена. Но все дело в том, что такие коллизии не легко обнаружить, и они связаны не только со специфической мультипрограммной организацией. Аналогичные ситуации могут возникнуть при использовании монитора или аппарата рандеву (см. гл. 6).

Подводя итог, можно сказать, что абстрактный подход к определению классов параметров представляется на первый взгляд достаточно удобным в теоретическом отношении. Он не требует от программиста рассмотрения передачи параметров в терминах ее реализации и не заставляет его учитывать форму представления объектов в машине. С другой стороны, этот подход связан с возникновением некоторых зависящих от реализации и чреватых ошибками ситуаций, которые могут оказаться весьма трудными для разрешения.

Определение классов параметров в терминах их реализации не ставит рассмотренных выше проблем, но если мы хотим сохранить ту же гибкость и надежность, которая присуща абстрактным классам, то придется реализовывать шесть различных классов: копировать-в, копировать-из, копировать-в-из, ссылаться-в, ссылаться-из, ссылаться-в-из. На практике такое множество классов параметров не приемлемо из-за его сложности. Кроме того, такое решение очень отрицательно повлияет на мобильность. Например, программист может выбрать ссылочный класс для передачи логического значения на машине, у которой маленький размер слова; при переносе программы на большую машину, где обычно логические значения упаковываются, ссылочный класс пришлось бы заменить на класс копирования.

На практике в проект языка придется ввести два или три из перечисленных классов, соглашаясь в отдельных случаях на некоторые ограничения и потерю эффективности. Приемлем выбор классов копировать-в (**const**), ссылаться-в-из (**var**) и копировать-из (**out**). Эти три класса обеспечивают такую же

гибкость, как и абстрактные классы, но приводят к некоторой потере надежности. Например, большой массив, передаваемый в процедуру лишь в качестве вводного, должен передаваться по классу **var**, и компилятор поэтому не смог бы обнаружить (ошибочное) присваивание этому массиву внутри процедуры. Кроме того, появилась бы некоторая потеря ясности и легкости использования при желании работать в классе копировать-в-из, так как это потребовало бы ввода значения с помощью параметра **const** и возврата значения с помощью другого параметра **out.**

В заключение отметим, что определение классов параметров связано с многочисленными трудностями, и сделать идеальный выбор не представляется возможным. В данной книге мы используем в примерах абстрактные классы **in, out** и **inout,** но это не означает, что мы предпочитаем выбранный подход. Для практических языков имеется тенденция выбора реализационно-ориентированного подхода с его неизбежными ограничениями и. потерей надежности (см. разд. 4.6).

4.4.3. Правила области действия

Как было упомянуто в разд. 4.4.1, общее определение процедур позволяет вставлять описание одних процедур в описание других процедур. Если в дальнейшем мы собираемся применять классические правила именной области действия для блочной структуры, вложенная процедура получает полную возможность обращаться к именам, описанным во внешних блоках. Поэтому в следующем примере все имена, описанные в процедуре *один* и процедуре *два,* доступны во второй процедуре:

```
procedure один;
   type статус = (вкл, выкл);
   const размер =100;
   var x, y : целый;
   procedure два;
     var s: статус;
   begin
      {размер, x, y, s здесь доступны}
   end;
begin
   {x, y, s здесь доступны}
end;
```

Хотя и безусловно верно, что возможность доступа к нелокальной переменной при случае может упростить разработку программы, на практике этой возможностью пользоваться не нужно (главным образом потому, что это сильно вредит ясности программы). Спецификация параметров процедуры ясно опи-

сывает ее интерфейс с внешней средой. Если к переменным во внешнем блоке обращаются непосредственно, то спецификации интерфейса становятся неполными. Другой, менее очевидной трудностью является то, что доступ к нелокальной переменной часто будет сопровождаться большими издержками времени работы программы, чем при передаче переменной в виде параметра.

Если согласиться, что нелокальный доступ, как правило, не желателен, но при случае полезен, то нужно признать необходимость обеспечения в языке некоторого метода управления именными областями действия, обеспечить такое управление можно несколькими способами. Чтобы проиллюстрировать эти возможности, рассмотрим здесь две из них.

Первый метод обеспечивает управление именной областью действия, разрешая помещать после заголовка процедуры оператор **use**. Оператор **use** перечисляет имена всех внешних идентификаторов, на которые можно производить ссылки внутри процедуры. Например, приведенную выше процедуру *два* можно переписать следующим образом:

```
procedure два;
  use статус, размер, x;
  var s:статус;
  begin
    {x, s, размер здесь доступны}
  end
```

В рассматриваемом случае оператор **use** делает имена *статус, размер* и *x* видимыми в процедуре, оставляя переменную *y* недоступной. Можно предположить, что обеспечение видимости имени типа *статус* автоматически делает видимыми также имена *вкл* и *вык*.

Оператор **use** обеспечивает простое и эффективное средство управления прозрачностью границ процедур для имен программы. Возможны различные вариации этого метода. Во-первых, если разрешить опускать оператор **use**, то его отсутствие можно интерпретировать либо как разрешение обращаться ко всем именам, либо как запрещение обращаться ко всем именам программы. У первой интерпретации есть то преимущество, что отпадает необходимость специфицировать очень длинные операторные списки: умолчание инициирует полную видимость. Впрочем, здесь есть опасность того, что программисты станут предпочитать вообще обходиться без операторов **use**. Поэтому следует, по-видимому, предпочесть вторую интерпретацию умолчания. Размер списков **use** можно уменьшить, ограничив управление видимостью только именами переменных, т. е. считать, что типы, константы и т. д. видимы всегда.

Второй метод управления именной областью действия состоит в связывании управления с каждой переменной в момент ее описания. По умолчанию считается, что во вложенном блоке все внешние имена недоступны. Имя может быть сделано видимым при связывании его со спецификатором **pervasive** (проникающий). Например, процедуры *один* и *два* можно записать следующим образом:

```
procedure один;
   type статус = (вкл, вык) pervasive;
   const размер = 100 pervasive;
   var x : целый pervasive;
        y: целый;
procedure два;
        var s: статус
    begin
      {x, s, размер здесь доступны}
   end;
 begin
   …
 end;
```

Специфицируя переменные *статус, размер* и *х как* проникающие переменные, мы получаем тот же результат, что и при использовании оператора **use** в предыдущем примере. Преимущество этого подхода состоит в том, что при необходимости передать имя через несколько уровней вложенности в глубоко вложенную процедуру достаточно объявить его проникающим. При использовании же первого подхода для передачи имени через очередные уровни вложенности нужно у каждой границы процедуры поместить оператор **use**. Как и в первом методе, количество употреблений спецификаторов **pervasive** можно уменьшить, сделав автоматически проникающими имена типов, констант и т. д.

Эти два метода управления именной областью действия показывают, как описания процедур можно сделать более надежными и ясными, требуя явной спецификации для нелокальных имен, на которые нужно делать ссылки. Такие механизмы просты для реализации и не уменьшают производительности программы. Отсюда следует сделать вывод, что включение такой особенности в язык реального времени может оказаться вполне оправданным.

4.4.4. Аспекты реализации

Реализация процедур в языке с блочной структурой описана в большинстве учебников по разработке компиляторов. Особенно четко это изложено в работе (Рол, 1975). Здесь мы рас-

смотрим лишь те аспекты, которые влияют на разработку языка.

При входе в процедуру для нее в стеке должно быть выделено место. Это место называется стековым отрезком. Но в отличие от простых блоков порядок входов и выходов из процедур во время компиляции предсказать нельзя, поэтому внутри каждого стекового отрезка должна содержаться некоторая «хозяйственная» информация. Эта информация касается адресов возврата, динамического и лексического контекстов. Динамический контекст регистрирует порядок, в котором вызываются процедуры, так что при выходе из каждой процедуры стек можно возвратить в его предыдущее состояние. Для этого обычно заводят указатель на стековый отрезок вызывающей процедуры. Лексический контекст специфицирует, какие из имеющихся в стеке стековых отрезков доступны для текущей процедуры при учете лексической вложенности описания процедуры. На больших машинах лексический контекст регистрируется реализованным аппаратно механизмом, который называется дисплеем. На меньших машинах он регистрируется указателем на стековый отрезок первой активной лексически объемлющей процедуры. Обращения к нелокальным переменным, таким образом, осуществляются косвенно через цепочку указателей до искомого стекового отрезка.

Из этого краткого описания ясно, что общая реализация процедур на малых машинах без специальной аппаратной поддержки может существенно увеличить время работы программы. Это связано как с ведением хозяйственной информации при вызове процедуры и выходе из нее, так и с обращением к переменным (особенно к нелокальным переменным) внутри процедуры. По этой причине многие проекты прежних языков реального времени предлагают процедуры лишь в сильно ограниченных формулировках. Например, при запрещении описаний вложенных процедур не требуется поддерживать лексический контекст и обеспечивать дорогостоящий доступ к нелокальным переменным. Ограничение типа локальных процедурных переменных простыми типами дает возможность обращаться ко всем локальным процедурным переменным командами с однократно индексируемыми адресами. Запрещение рекурсивного обращения к процедурам может позволить выполнить эффективную оптимизацию при распределении памяти для локальных переменных. Однако, несмотря на привлекательность таких ограничений в смысле гарантии высокой производительности программ, имеются причины, по которым в современных проектах языков реального времени не следует идти на эти. ограничения.

Во-первых, компилятор может оптимизировать вызовы процедур таким образом, что будет фиксироваться лишь по существу необходимая хозяйственная информация. Поэтому, например, если программа написана так, что в ней нет никаких обращений к нелокальным переменным внутри вложенных процедур, то никаких дополнительных издержек времени работы программы не будет. В этой связи следует заметить, что включение в проект операторов **use** значительно облегчает обнаружение аналогичных условий. Во-вторых, современная тенденция в проектировании машинного оборудования заключается в разработке такого множества машинных команд, которые, в частности, поддерживают реализацию языков высокого уровня, и особенно в области реализации процедур. Поэтому было бы жаль ограничивать проект языка до такой степени, что эта машинная поддержка окажется неиспользованной. В-третьих, что, по-видимому, самое важное, понятие процедуры, которая имеет одинаковые с главной программой права и может содержать вложенные процедуры с такими же правами, является важнейшим средством построения программ с использованием методологии разработки на основе последовательного уточнения. Если использование таких общих процедур ведет к резкому уменьшению производительности программ, то программу можно оптимизировать, отказавшись от дорогих особенностей в конкретных критических областях. Однако подобной возможностью должен полностью распоряжаться программист, проект языка не должен содержать этой особенности в качестве общего требования, ограничивающего не только структуру критичных частей, но и остальные некритичные части.

4.5. ФУНКЦИИ И ОПЕРАЦИИ

4.5.1. Функции

Механизм процедур, описанный в предыдущем разделе, является самой общей возможностью вычисления множества выходных значений по множеству входных значений. При этом вычисление может сопровождаться побочными эффектами. Хотя процедура сама по себе способна адекватно представлять любой вид вычислений, ее употребление в некоторых ситуациях может оказаться неудобным из-за громоздкости. Например, при использовании процедуры для вычисления единственного значения при арифметической обработке нужно описывать дополнительные временные локальные переменные для хранения результата работы вызванной процедуры. Ввиду этого базовый механизм процедур обычно расширяется за счет введения очень похожей конструкции, называемой функцией.

Функция — это специальная форма процедуры, предназначенная для вычисления одного значения. Для вызова функции в отличие от процедуры ее имя нужно написать в виде компоненты выражения. Вычисленное функцией имя связывается затем с именем самой функции, что позволяет обходиться без промежуточных локальных переменных, передающих выходное значение. В качестве примера рассмотрим следующую функцию *max,* возвращающую значение наибольшего из своих параметров:

```
function max(i, j: целый): целый;
begin
   if i > j then
      return (i)
   else
     return(i)
   end if
 end;
```

Примером использования этой процедуры является оператор $k :=$ max (3,4) + *7,* при выполнении которого *k* присвоится значение 11.

Разработка функций тесно связана с разработкой процедур. Синтаксически они идентичны; однако у функции нужно специфицировать тип результата. Тело функции идентично телу процедуры за исключением того, что внутри тела функции нужно специфицировать значение функции. В рассматриваемом примере это сделано с помощью введения оператора **return**. Основное отличие формулировок процедур и функций связано с необходимостью ограничить возможное взаимодействие функции со средой.

С операционной точки зрения функция в языке программирования аналогична функции в математике. Поэтому важно сохранить интуитивный смысл обращения к функции в рамках оператора. В частности, естественно ожидать выполнения следующих отношений:

$$f + f = 2*f$$
$$f + g = g + f$$

где f и g - функции. Такие отношения могут быть справедливыми только в языке, гарантирующем отсутствие у функций побочных эффектов, т. е. отсутствие какой-либо модификации функцией среды во время своего исполнения. Наряду с тем что такую программу легче понимать, это также предоставляет компилятору больше свободы в выборе порядка вычисления компонент выражения. Поэтому правило «отсутствия побочных эффектов» должно применяться к функциям, по-видимому, всегда. Правило отсутствия побочных эффектов требует, чтобы все

входные параметры принадлежали только классу **in**, функция не могла модифицировать глобальные переменные и функция не могла выполнять операции ввода-вывода. Из первого требования следует, что явное указание класса в описании функции не нужно (см. выше процедуру *max*). Из второго и третьего требований следует, что это правило должно применяться транзитивно ко всем вызовам подпрограмм, которые делает функция.

Определение функции без побочных эффектов сводится по существу к спецификации чистой функции. Наличие чистых функций является очень полезной особенностью языка программирования. Кроме того, они допускают очень эффективную реализацию в связи с ограничением доступа к нелокальным переменным. Функция может прочесть нелокальную переменную; но так как компилятору известно, что она не сможет ее модифицировать, он имеет возможность легко оптимизировать такие обращения, образовав локальную копию переменной при входе в функцию. Однако следует заметить, что реализация правила отсутствия побочных эффектов делает компилятор более сложным.

В заключение отметим, что в практическом языке программирования следует оставить лазейку, позволяющую программисту в некоторых специальных случаях писать нечистые функции. Например, во время отладки может потребоваться так инструментировать функцию (например, введя трассировочную распечатку), чтобы можно было управлять ее работой. Обеспечить эту возможность можно было бы с помощью компиляторной директивы **impure** (нечистая), включаемой в описание функции.

4.5.2. Совмещение

Если предполагается, что к именам подпрограмм (т.е. функциям и процедурам) применяются обычные правила области действия при блочной структуре, то описание подпрограммы *x* внутри вложенного блока автоматически спрячет любую другую подпрограмму с тем же именем *x,* описанную во внешнем блоке, и любая попытка описать дважды подпрограмму *x* в одном и том же блоке приведет к выдаче компилятором сообщения об ошибке. Хотя использование общего правила для всех идентификаторов (переменных, типов, функций и т. д.) просто выучить программисту и просто реализовать компилятору, имеются причины, по которым следовало бы разрешить одновременное существование более чем одного определения некой подпрограммы. Механизм, с помощью которого это достигается, называется совмещением.

Если проект языка разрешает совмещение подпрограмм, то обычные правила именной области действия не применимы к именам подпрограмм. Вместо них используются следующие правила:

(i) Два определения подпрограммы *x* не различимы, если у них одинаковое число параметров и все соответствующие параметры одного и того же типа. Если подпрограмма является функцией, то типы результатов должны быть также идентичны.

(ii) Если подпрограмма *x* описывается в той же самой области действия, что и существующая подпрограмма *x,* то при неразличимости нового и существующего определений либо временно отменяется существующее определение, если оно находится во внешнем блоке, либо компилятор выдает сообщение об ошибке. В противном случае новое определение совмещает *x,* и оба определения становятся видимыми в текущей области.

При вызове совмещенной подпрограммы компилятор выбирает подходящее определение, анализируя задаваемые обращением типы параметров.

Имеются три основных аргумента, оправдывающие включение механизма совмещения в язык реального времени. Во-первых, совмещение используется фактически во всех языках программирования при определении стандартных функций. Например, часто вводится функция *abs,* вырабатывающая абсолютное значение своего аргумента. Если аргумент целого типа, то возвращаемый результат тоже целого типа, если же тип аргумента вещественный, то и тип результата вещественный. Другими словами, abs является совмещенной функцией с двумя определениями:

function *abs*(*i* : *целый*) : *целый*;
function *abs*(*x* : *плавающий*) : *плавающий*;

Мы, конечно, хотим, чтобы все поддерживающее программное обеспечение для языка программирования писалось на самом языке, включая библиотеки подпрограмм. Это возможно лишь при обеспечении языком совмещения.

Во-вторых, язык должен быть расширяемым, и, в частности, все определенные пользователем типы должны иметь такие же права, как и предопределенные типы. Но если нет возможности воспользоваться совмещением, то это требование может быть удовлетворено лишь частично. Например, предположим, что в языке имеется предопределенная (совмещенная) процедура *писать* для выдачи значений всех стандартных типов. Тогда при введении определенного пользователем типа *комплексный* нужно было бы определить и соответствующую операцию вы-

вода, имеющую уникальное имя, например

procedure *писать_комплекс*(**in** *c* : *комплексный*);

что вряд ли соответствует принципу «равных прав» для определенных пользователем типов. При обеспечении совмещения можно дать дополнительное к процедуре *писать* определение, т. е.

procedure *писать*(**in** *c* : *комплексный*);

В-третьих, за исключением некоторых рассмотренных ниже случаев, совмещение является фактически «прозрачным» для имен. Поэтому наивный пользователь может при желании считать, что справедливы классические правила именной области действия. Другими словами, программа, написанная в предположении классических правил именной области действия для блочной структуры, будет работать так же надежно и с правилами совмещения (обратное может оказаться и неверным). Поэтому совмещение можно считать специфической возможностью, которую при желании можно спокойно проигнорировать.

В дополнение к сделанным выше замечаниям следует также сказать, что если мы хотим, чтобы язык поддерживал определяемые пользователем операции, то требуемый для поддержания совмещения механизм должен быть заранее встроен в компилятор. Следовательно, в свете рассмотренного третьего аргумента кажется логичным распространить понятие совмещения также на процедуры и функции.

Эти аргументы в пользу совмещения нужно, однако, сопоставить с рядом его недостатков. Во-первых, введение совмещения значительно увеличивает сложность компилятора и может сильно уменьшить скорость компиляции. Чтобы проиллюстрировать трудности, связанные с реализацией совмещения, рассмотрим следующую ситуацию. Предположим, что описаны следующие совмещенные функции:

(a) **function** *f*(*n* : *цел*1) : *цел*1;
(b) **function** *f*(*n* : *цел*1) : *цел*2;
(c) **function** *g*(*n* : *цел*1) : *цел*1;
(d) **function** *g*(*n* : *цел*2) : *цел*1;

где *цел*1 и *цел*2 производные целые типы. Пусть у переменной t тип *цел*1. Рассмотрим, что должен делать компилятор, чтобы выбрать подходящие определения для *f* и *g* в следующих четырех операторах присваивания:

(1) $i := f(i)$
(2) $i := g(i)$
(3) $i := f(g(i))$
(4) $i := g(f(i))$

В операторе (1) тип параметров функции *f* не дает информации для решения, какое из определений *f* имелось в виду. Компилятор должен поэтому просматривать дерево грамматического разбора выражения сверху вниз, используя ожидаемый тип результата *цел*1, чтобы выбрать для *f* определение (а). В операторе (2), однако, компилятор должен просматривать выражение снизу вверх, используя тип параметров *g*, чтобы выбрать для *g* определение (с). В более сложном случае, таком, как оператор (3), компилятор должен просматривать дерево грамматического разбора сначала сверху вниз, а затем снизу вверх. При просмотре сверху вниз компилятор может сразу же выбрать для *f* определение (а), но не может разрешить ситуацию с *g* до тех пор, пока не будет определен тип ее параметров. Затем, возвращаясь обратно по дереву, компилятор может выбрать определение (с). В более сложных выражениях компилятор может оказаться вынужденным просматривать дерево вверх и вниз несколько раз, прежде чем выбрать правильные определения для всех совмещенных функций. При разработке компилятора нужно обратить внимание на то, чтобы при попытке разрешить двусмысленные выражения компилятор не зациклился. Примером такой ситуации является оператор (4), где нельзя разрешить неопределенность ни для *f*, ни для *g*. В дополнение к рассмотренным примерам следует указать на трудности, связанные с присутствием совмещенных литералов (например, тип 1 может быть либо *цел*1, либо *цел*2), что может также привести к двусмысленности выражения. Например, $i := g(1)$ является неразрешимой двусмысленной записью. Чтобы сделать этот оператор однозначным, нужно дописать квалификатор типа, например, *i* := *g*(*цел*2(1)).

Вторым недостатком совмещения является то, что оно, безусловно, связано с некоторой потерей надежности, так как для выбора определений нужна информация о типе, которая теперь не может использоваться для проверок соответствия типов. Поэтому при наличии нескольких определений подпрограммы может произойти обращение к этой подпрограмме с неправильным типом, что приведет к совершенно законному обращению к другому определению подпрограммы, не к тому, которое имелось в виду. Отсюда следует, что хотя совмещение на самом деле не видно пользователю, который предпочитает рассуждать в терминах классических правил области действия в случае правильных программ, оно достаточно заметно в ситуациях, где имеются ошибки. По этой причине будет, по-видимому, лучше рассматривать совмещение как расширение классических правил области действия, а не как их замену. Из этого следует, что перед совмещенным определением нужно помещать подходящее ключевое слово, чтобы отличать его от нормального

описания, которое покрывает все существующие использования того же самого имени.

В заключение отметим, что наличие совмещения в языке реального времени имеет свои положительные и отрицательные стороны. К сожалению, мы обладаем слишком малым опытом его использования, чтобы высказать более определенное суждение. Как было замечено в разд. 4.6, ни один из широко применяемых языков реального времени совмещения не предоставляет. Впрочем оно есть в языке Ада; прежде чем высказать окончательное суждение о желательности совмещения в языке, подождем, что покажет опыт эксплуатации Ады.

4.5.3. Операции

Возможность описывать в языке задаваемые пользователем операции позволяет пользователю расширить множество стандартных операций, включая определенные им самим типы. Эта возможность не очень существенна, так как требуемые операции могли бы быть выражены в форме функций. Впрочем, если стремиться к осуществлению принципа «равных прав для определенных пользователем типов», то определенные пользователем операции безусловно требуются. Вопрос о том, стоит ли реализовывать этот принцип, увеличивая при этом сложность языка, является открытым. Определенные пользователем операции неявно требуют механизма совмещения, поэтому все высказанные в предыдущем разделе соображения имеют прямое отношение к обсуждаемому вопросу. Здесь мы рассматриваем разработку и использование такой возможности определения операций. К вопросу о необходимости включения ее в язык реального времени мы вернемся снова в гл. 9.

Возможность общего определения операций предоставила бы пользователю средство переопределения существующих операций, совмещения существующих операций, введения совершенно новых операций и спецификаций для операций произвольных приоритетов. Такой общий механизм, однако, предоставляет несколько чрезмерную гибкость, что неизбежно приводит к порождению малопонятных программ. И конечно, реализация оказалась бы дорогой. Если заботиться только об обеспечении базового требования предоставления определенных пользователем типам одинаковых с предопределенными типами прав, то достаточно предоставить только совмещение существующих операций, считая все приоритеты предопределенными и фиксированными. Это упрощает реализацию и сохраняет ясность и смысл программ. Руководящим принципом здесь является сохранение, где это возможно, интуитивного смысла операции. Если разрешается только совмещение существующих операций, то язык может навязать некоторые условия их использования. Напри-

мер, если совмещается операция равенства « = », то требуется, чтобы типы операндов были одинаковыми, а тип результата логическим.

Синтаксически описание операции может быть идентичным описанию функции с отличием только в ключевом слове. В качестве примера предположим, что требуются комплексные числа в языке, в котором нет соответствующего предопределенного типа. Подходящее определение типа *комплексный* может быть записано в следующем виде:

```
type комплексный = record
                       re, im: плавающий
                   end
```

Затем можно определить различные арифметические операции, совмещая стандартные символы «+», «-», «*» и «/». Например, операция «+» могла бы быть определена так:

```
operator + (a, b : комплексный) : комплексный;
var c: комплексный;
begin
   c.re := a.re + b.re;
    c.im := a.im + b.im;
   return(c)
end;
```

где параметры *a* и *b* обозначают соответственно левые и правые операнды.

Совмещение арифметических операций в общем случае не должно противоречить основным аксиомам этих операций. Типы операндов и результата должны быть теми же самыми, и должны быть сохранены их коммутативные и ассоциативные свойства (подробное обсуждение этого вопроса можно найти в работе (Пайл, 1980)). На самом деле может оказаться выгодным попросить программиста явно специфицировать с помощью подходящих указаний компилятору, является ли операция коммутативной и (или) ассоциативной. Например, при расширении приведенного выше набора арифметических операций за счет операции, у которой значение одного операнда вещественное, а другого комплексное, потребовалось бы совмещение такого вида:

```
operator + (a : плавающий; b : комплексный): комплексный;
operator -f (a : комплексный; b : плавающий): комплексный;
```

Однако если компилятору известно, что «+» коммутативная операция, то достаточно привести только одно из этих определений, т. е.

```
commutative operator + (a : плавающий;
                        b : комплексный): комплексный;
```

после чего для компилятора такие выражения, как 3.0 + *c* и *c* + 3.0, эквивалентны.

И наконец, имеется специальный случай, когда типы операндов и результата для «*» и «/» не одинаковы по определению. То же самое бывает тогда, когда производные типы используются для представления размерных единиц. Произведение двух типов *T*1 и *T*2 приводит к появлению третьего типа *T*3, т. е.

operator*(*a* :*T*1; *b* : T2) : T3;

Для согласования соответствующая операция «/» должна всегда определяться как

operator/ (*a* : *T*3; *b* : *T*1) : *T*2;

и (или)

operator/ (*a* : *T*3; *b* : *T*1) : *T*1;

Операция «*» может как быть, так и не быть коммутативной. Пример таких производных типов приводился в разд. 2.2.5, где вычислялась площадь a как произведение двух длин *x* и *y*, т. е.

a := *x***y*

Подходящим совмещением «*» в этом случае может быть

```
commutative operator*(x, y : длина): площадь;
begin
   return (площадь (плавающий (x)*плавающий (y)))
end;
```

где тело описания состоит всего лишь из соответствующего преобразования типов.

Использование производных типов для представления размерных единиц требует от программиста определенных усилий. Если базовый синтаксис языка позволяет строить выражения вида

*e***f***g***h**....

то мы не можем их использовать в случае, когда типы компонент размерны, так как в этом случае такие формы бессмысленны. Например, запись

*x***y***z*

где *x, y* и z типа *длина,* была бы не закончена даже в случае ее синтаксической правильности (если, конечно, не был введен новый тип *объем* с соответствующим дальнейшим совмещением «*»).

Подводя итог, можно сказать, что возможность описывать определенные пользователем операции с помощью совмещения предопределенных операций расширяет язык в том смысле, что определенным пользователем типам можно предоставить равные права с существующими предопределенными типами. За предоставление этой возможности мы расплачиваемся необходимостью введения механизма совмещения, что обсуждалось в предыдущем разделе. Эта плата может оказаться довольно значительной в смысле усложнения компилятора и увеличения времени компиляции. Впрочем, стоит заметить, что часть расходов может окупиться при эксплуатации потенциала расширения языка. Если в некоем языке описывать операции не разрешается, то, возможно, придется обеспечивать в нем такие типы, как *строчный, комплексный* и т. д., в качестве предопределенных типов. От расширяемого языка требуется обеспечивать только базовые примитивные типы; дополнительные типы и операции вводит сам пользователь по мере надобности.

4.6. ПРОГРАММНЫЕ СТРУКТУРЫ В ПРАКТИЧЕСКИХ ЯЗЫКАХ

Наиболее разработанным средством структурирования программ является, по-видимому, механизм операторного управления. В языке RTL/2 (Барнес, 1976 и гл. 11) это сделано особенно хорошо и удовлетворяет предложенным в этой главе критериям. Кроме стандартных структур управления испытывались также и некоторые необычные конструкции. В языке Модула (Вирт, 1977 и гл. 12), например, имеется разновидность оператора выхода из цикла, содержащая блок операторов, который нужно выполнить до фактического выхода из цикла при истинности условия выхода. Интересно отметить, что Вирт, разработавший языки Паскаль и Модула, при разработке Модулы исправил неточности, которые были замечены в структурах управления языка Паскаль. Структуры управления языка Ада (см. гл. 13) близко соответствуют конструкциям из разд. 4.2.3.

Механизмов процедур много, и они разные. На одном полюсе находится язык RTL/2, механизмы которого очень просты и ограничены. Процедуры RTL/2 не могут быть вложенными, а в качестве локальных переменных можно описывать переменные только простых типов. Имеется только один параметрический класс, являющийся классом копировать-в, причем параметры должны быть простых типов. Впрочем, в языке RTL/2 имеются ссылочные типы, примерно такие, как в языке Алгол 68, которые обеспечивают возврат результатов из процедур и позволяют обрабатывать большие структуры данных. На другом полюсе находится язык Ада, обеспечивающий вложенность процедур и функций любой глубины. Имеются абстрактные

классы **in, out** и **inout,** причем на тины параметров не накладывается никаких ограничений.

Фактически во всех языках применяются для описаний вложенных процедур классические правила именной области действия. Но в некоторых имеются дополнительные возможности управления областью действия. Модула, например, позволяет приводить дополнительно список использования; при этом видимыми становятся только те внешние имена, которые перечислены в списке. Пустой список использования показывает, что видимых внешних имен нет, а отсутствие списка использования эквивалентно тому, что видимы все внешние имена. Язык Redl (Нестор, 1978) предоставляет управление областью действия как в форме импортного списка (то же, что и список использования), так и в форме директив проникаемости (pervasive), расширяющих видимость индивидуальных имен. По умолчанию область подпрограмм в языке Redl закрыта, т. е. если внешние имена не импортируются и не являются проникающими, то они невидимы. И наоборот, в языке Ада используются правила открытой области действия и не имеется никакого механизма их ограничения.

Очень мало языков разрешают совмещение имен и описание операций. Алгол 68 (Ван Вейнгаарден и др., 1975) является, по-видимому, единственным широко используемым языком, который предоставляет такие возможности. Совмещение ограничивается только операциями, но возможность определения операций достаточно общая. Могут вводиться символы новых операций и устанавливаться произвольные приоритеты для новых и существующих операций. Ада обеспечивает совмещение для любых подпрограмм, а операции можно совмещать аналогично тому, как мы описали в предыдущем разделе. Ада не требует, чтобы совмещение было явно указано с помощью ключевого слова, вместо этого работает базовый механизм для областей действия имен подпрограмм. В противоположность этому в языке Redl для спецификации свойств подпрограмм имеется целое множество ключевых слов/директив, таких, как **overload, commutative, associative, recursive и re-entrant.** Особенно интересной в языке Redl является директива **safe.** Все подпрограммы, которые вызываются из функции, должны иметь спецификатор **safe** (надежный), что означает отсутствие у них побочных эффектов. Компилятор проверяет, что у каждой подпрограммы, специфицированной как надежная, действительно нет побочных эффектов, и затем отмечает в своей лексической таблице, что эта подпрограмма надежная. Это очень упрощает проверку функции на чистоту, так как для такой проверки нужно только удостовериться, что все внешние обращения делаются к надежным подпрограммам.

Глава 5. МЕХАНИЗМЫ АБСТРАКЦИИ

5.1. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В предыдущих главах мы обращали особое внимание на метод разработки программ с помощью процесса последовательного уточнения. Требуется, чтобы этот метод был применим как к разработке структур данных, так и к разработке программных структур. Поэтому выбирались такие формы языковых особенностей для выражения этих структур, которые бы соответствовали данному требованию. Например, компоненты записи могут быть произвольного типа, что позволяет уточнять сложные структуры данных на различных уровнях. Аналогично этому, процедуры можно определять в терминах других вложенных процедур, что также позволяет уточнять сложные действия на нескольких уровнях. Отсюда следует, что такие конструкции структурных данных, как записи и массивы, являются в некотором смысле средством абстрагирования для спецификаций данных, а процедуры и функции — средством абстрагирования для спецификаций действий.

В области разработки программного обеспечения усиливается тенденция унифицировать двойственное уточнение данных и действий с помощью проектирования программ с использованием понятия абстрактного типа данных (см., например, Уэлш и Мак-Киг, 1980). Этот метод по существу сводится к спецификации проекта программы на некотором уровне уточнения в терминах множества абстрактных типов данных и связанных с ним операций. Затем на следующем уровне уточнения будут уточняться абстрактные типы предыдущего уровня и будут определяться операции над ними, возможно, в терминах следующих (чуть менее) абстрактных типов данных. В частности, например, программа, которой нужно манипулировать с объектами, представляющими время суток, могла бы на уровне *N* постулировать существование абстрактного типа *время* следующим образом:

Abstract Type Specification
Имя типа : *время*
Операции : **procedure** *восстан*(**inout** *t* : *время*);
procedure *приращ*(**inout** *t*: *время*);
function *слож*(*t*1, *t*2 : *время*) : *время*;
.
.
.
и т. д.

Тогда вариант программы на уровне *N* мог бы содержать описания объектов типа *время* так, как будто *время* является предопределенным типом, например

```
var время_теперь, время_потом : время;
```

Оперировать с этими объектами можно с помощью специфицированных операций, например

```
приращ(время_теперь)
```

может служить для увеличения *время_теперь* на одну секунду. Самое интересное здесь - это то, что разработка на уровне *N* никак не учитывает внутреннюю структуру объектов типа *время* и процесс выполнения операций над ними. Но она, и в этом все дело, не должна этим заниматься, иначе мы подвергнем опасности целостность объектов типа *время.*

На уровне *N* + 1 абстрактный тип *время* может быть определен в терминах имеющихся языковых конструкций. Например,

```
type время = record
                 час : целый range 0..23;
                 мин, сек : целый range 0..59
             end;
procedure восстан(inout t : время);
begin
   t.час:=0; t.мин:=0; t.сек:=0
end;
procedure приращ (inout t: время);
begin
   .
   .
   .
```

Но возникает вопрос, куда нужно поместить эти описания абстрактного типа *время.* Ясно, что они должны находиться в области действия всех программных единиц, которые хотят использовать тип *время.* Из этого следует, что их нужно поместить в описательную часть самой внешней программной единицы. Предположим, что эта единица является процедурой с именем *использует_время.* Ее структура будет иметь следующий вид:

```
procedure использует_время;
.
.
.
type время = record...
procedure восстан...
procedure приращ...
function слож... и т. д.
var время_теперь, время_потом : время
{другие описания для использует_время}
```

```
begin
.
.
.
end;
```

Однако эта структура не отражает понятия времени как абстрактного типа данных. Более того, внутренняя структура типа *время* в процедуре *использует_время* видима везде, тогда как она должна быть видима только для тех процедур, которым разрешено работать с типом *время.*

Эти проблемы можно разрешить только путем введения нового вида программных единиц, которые позволяли бы объединять совокупности объектов, таких, как рассмотренный тип *время,* и операции над ними, чтобы стало возможно осуществлять строгое управление доступом из внешней среды. Эта новая программная единица будет называться модулем.

Модуль состоит из двух частей: спецификации интерфейса и приватной реализационной части. Например, абстрактный тип *время* может быть введен с помощью модуля следующим образом (применяемую нотацию мы разработаем подробнее в дальнейших разделах):

```
module тип_время;
  define type время;
        procedure восстан(inout t : время);
        procedure приращ(inout  t: время);
        function слож(t1, t2 : время) : время;
        .
        .
        .
        и т. д.

 private
   {определения для время, восстан и т. д.}
 end module тип_время;
```

Названные в списке **define** объекты во внешней среде доступны, тогда как все детали реализации в части **private** спрятаны. В частности, имя типа *время* известно вне модуля, а его внутренняя структура не известна.

Модуль может быть помещен в программе везде, где может быть помещена процедура. Поэтому, используя введенный модуль, мы получаем такую структуру процедуры *использует_время*:

```
procedure использует_время;
 module тип_время;
    .
    .
    .
```

```
    end module тип_время;
    var время_теперь, время_потом : время;
     .
     .
  begin  end;
```

Теперь в процедуре *использует_время* видимы только те объекты, которые перечислены в списке **define** в модуле *тип_время.*

Конструкцию модуля можно сравнить с неким забором вокруг описанных объектов. Спецификация интерфейса — это окно в заборе. Видимы только те объекты, которые описаны в окне, и только они доступны из внешней среды. Поэтому конструкция модуля обеспечивает четкую реализацию понятия абстрактных типов данных простым и надежным способом. Модуль гарантирует целостность описанных абстрактным типом данных объектов, не давая внешней среде никакой информации о внутренней структуре типа, и поэтому делает невозможным выбор или модификацию какой бы то ни было компоненты, кроме как с помощью предоставленных операций.

5.7. ЧТО ЕЩЕ СВЯЗАНО С ВВЕДЕНИЕМ ПОНЯТИЯ МОДУЛЯ

Понятие модуля было введено в предыдущем разделе в качестве простого механизма реализации абстрактных типов данных. Но с помощью модуля можно решать и более трудные задачи. На самом деле модули должны быть первичными строительными блоками в современных языках реального времени. Модуль дает возможность реализации программных подсистем, специфицируя интерфейс с этими подсистемами и охраняя эти подсистемы от вмешательства в их работу со стороны внешних «пользователей». Фактический вид подсистемы может варьироваться. Это может быть реализация абстрактного типа, как мы только что рассмотрели, это может быть ресурс, такой, как буферная память или система управления файлами, или это может быть просто пакет полезно соотносящихся процедур. Итак, модуль есть средство «крупномасштабного программирования», и его основные достоинства проявляются при обуздании сложности современных систем реального времени.

В качестве примера использования модуля для реализации ресурса рассмотрим стек LIFO, который должен содержать максимум 100 объектов типа *элем.* Для обеспечения этого ре-

сурса можно написать модуль со следующей основной структурой:

```
module стек;
   define procedure втолкнуть(in x : элем);
          procedure вытолкнуть(out x : элем);
 private
   {описания для стека, например
     type идстк = целый range 0..100;
      var s : массив[1..100] of элем;
          sp : идстк;
   и описания для втолкнуть и вытолкнуть}
 end module стек;
```

Здесь мы ввели стек, который осуществляет интерфейс с «пользователем» с помощью всего двух обращений к процедурам *втолкнуть* и *вытолкнуть.* Реализация стека скрыта от пользователя. Он не знает, как работает, стек, и ему не требуется это знать. В действительности реализация стекового модуля позднее может быть изменена (возможно так, чтобы использовать динамическую память), но это никак не отразится на пользователе. Пользователь не может изменять значения указателя стека, так как не имеет к нему доступа, и поэтому не может испортить стек.

Пример со стеком показывает некоторые из основных преимуществ, которые дает использование модулей. В общем случае эти преимущества состоят в следующем:

(i) Модуль группирует логически соотносящиеся объекты в одну единицу, тем самым непосредственно реализуя соответствующее абстрактное понятие, предусмотренное в процессе разработки.

(ii) Спецификация интерфейса в точности определяет, какие возможности предоставляет модуль внешней среде. Это позволяет отделить приватную часть и поместить ее отдельно от текста главной программы, так что не относящиеся к делу реализационные подробности не затемняют основной идеи разработки.

(iii) Модуль помогает отлаживать программу. Ограниченный доступ к модулю исключает влияние ошибок во внешней среде на его функционирование. Поэтому систему можно строить методом снизу вверх, по очереди подключая каждый из отлаженных модулей. После присоединения отлаженного модуля можно считать, что все появляющиеся новые ошибки содержатся в новом модуле, а не в уже отлаженных. Другими словами, границы модуля не дают программистским ошибкам возможности распространяться по всей программе.

(iv) Модуль помогает строить очень большие системы в совместной работе группам программистов. Как только становятся определены спецификации интерфейса, каждый член группы может программировать различные подсистемы независимо от других членов группы. То, как реализован каждый модуль, не влияет на его интерфейс с внешней средой.

(v) Модуль помогает сопровождать программы. В отличие от процедур, которые могут взаимодействовать с внешней средой, модифицируя нелокальные переменные, модуль является совершенно закрытой областью (за исключением тех объектов, которые явно упомянуты в спецификации интерфейса). Поэтому модуль может произвольно модифицироваться, и при условии сохранения спецификаций интерфейса такие модификации не могут повредить внешней среде. Это резко отличает модульный подход от классической методологии разработки программ, которая базируется только на процедурах, где небольшое изменение в одной части программы может вызвать, цепочку изменений, распространяющуюся по всей программе.

Отметим в заключение, что идея модуля очень проста, но с помощью этой простой идеи мы получаем неожиданно хорошие результаты. По существу, это всего лишь текстуальная группировка соотносящихся друг с другом объектов в единицы с дополнительным уровнем управления именной областью действия. Как будет показано в разд. 5.4, модули не увеличивают время работы готовой программы, обработка модулей целиком осуществляется во время компиляции. Поэтому, принимая во внимание тот факт, что модули являются мощным средством структурирования программ, и учитывая их перечисленные выше достоинства, модули следует считать абсолютно необходимой конструкцией в проекте любого современного языка реального времени. Разработку этой конструкции мы рассмотрим теперь более подробно.

5.3. РАЗРАБОТКА МЕХАНИЗМА ОРГАНИЗАЦИИ МОДУЛЯ

5.3.1. Общая структура

Как было указано выше при рассмотрении неформального примера, модуль состоит из двух частей: из части спецификации интерфейса и приватной реализационной части. В общем случае спецификация интерфейса состоит из списка **devine** и списка **use**. В первой части специфицируются те объекты, которые описаны внутри модуля, но доступны для обращений из внешней среды. О таких объектах говорят, что они экспортируются из модуля. Во второй части специфицируются те объекты, которые описаны во внешней среде, но которые доступны

из модуля. О таких объектах говорят, что они импортируются в модуль. Реализационная часть имеет в точности такую же структуру, как и блок, т. е. в ней имеется описательная часть и операторное тело. Описательная часть содержит определения всех объектов, которые используются в модуле, включая и те, которые экспортируются. Операторная часть служит для инициализации модуля, она исполняется, когда появляется объемлющая программная единица.

Следующий пример иллюстрирует вид модуля и его интерфейс с внешней средой для случая процедуры. Пример представляет завершенную версию стекового модуля, введенного в предыдущем разделе.

```
 procedure использование_стека;
    type элем = целый range —2048..2047;
    module стек;
        use type элем;
       define procedure втолкнуть(in x : элем);
              procedure вытолкнуть(out x : элем) ;
    private
       type идет = целый range 0..100;
       var s : array [1..100] of элем;
           sp : идст;
       procedure втолкнуть(in x : элем);
       begin
          sp := sp + 1;  s[sp] := x
       end;
       procedure вытолкнуть(out x : элем);
       begin
          x:=s[sp]; sp:=sp - 1
       end;
 begin
       sp := 0
   end module стек;
   var x : элем;
begin
   .
   .
   .
   втолкнуть(1); вытолкнуть(x) и т. д.
   .
   .
   .
end;
```

В этом примере нужно обратить внимание на следующее. Во-первых, тип *элем* описан вне модуля *стек* и должен поэтому импортироваться в модуль, что достигается включением его в

список **use** (ср. с обсуждением управления областью действия для процедур в разд. 4.4.3). Во-вторых, внутри приватной реализационной части модуля описано пять объектов: тип *идет,* переменные *s* и *sp* и процедуры *втолкнуть* и *вытолкнуть.* Однако только два последних объекта экспортируются в процедуру *использование-стека,* остальные три объекта являются для модуля полностью приватными. Поэтому независимо от того, какие могут быть сделаны ошибки при разработке процедуры *использование-стека,* целостность самого стека как буфера LIFO не может быть нарушена (хотя заметим, что для простоты проверки переполнения/исчерпания стека были опущены). В-третьих, операторное тело модуля содержит один оператор для инициализации стекового указателя. Этот оператор будет выполняться каждый раз при создании процедуры *использование_стека,* т. е. каждый раз при ее вызове. В-четвертых, время жизни переменных *s* и *sp* такое же, как и время жизни объемлющей программной единицы. Поэтому при вызове процедуры *использование_стека* переменные *s* и *sp* создаются, а при ее завершении исчезают. И наконец, следует заметить, что для понимания того, как работает процедура *использование_стека,* включать в рассмотрение нужно только текст модульного интерфейса. Приватная часть не существенна и поэтому может быть опущена.

```
procedure использование_стека;
    type элем = целый range -2048..2047;
    module стек;
        use type элем;
        define procedure втолкнуть(in x: элем);
            procedure вытолкнуть(out x : элем);
    end module стек;
    var x: элем;
begin
        .
        .
        .
(втолкнуть(\): вытолкнуть(x) и т. д.
        .
        .
        .
end
```

Полностью стековый модуль может быть затем описан в программе позднее (или даже совершенно самостоятельно - см. разд. 9.6), чем достигается разделение описательной части процедуры *использование_стека* и не относящихся к делу реализационных деталей. Это особенно важно тогда, когда модули очень велики.

5.3.2. Спецификация интерфейса

Основные вопросы проектирования, на которые нужно обратить внимание при определении механизма модуля, относятся к спецификации интерфейса. Первый и самый главный вопрос, который нужно решить, - какие виды объектов разрешается пропускать через границы модуля. Если модули резервируются только лишь для реализации абстрактных типов данных и ресурсов, то достаточно пропускать только типы и процедуры. Впрочем, иногда может оказаться полезным иметь возможность пропускать также константы и переменные. В первом случае импортирование констант в модуль позволяет в некоторой степени производить параметризацию абстрактных типов, описанных внутри модуля. Например, приведенный ранее модуль мог бы импортировать константу *размер_стека,* описанную во внешней среде, для установки значения размера стека. Во втором случае экспортирование переменной может пригодиться в ситуациях, когда из соображений эффективности требуется минимизировать время обращения к процедуре. Но здесь, по-видимому, было бы разумно предположить, что такие экспортируемые переменные доступны только для чтения; это следует из нашего желания наилучшим образом обеспечить сохранность модуля. На практике экспортируемые переменные полезны лишь тогда, когда они простых типов. Экспортирование структурных переменных практически почти бесполезно, так как внутренние компоненты переменной недоступны никогда.

Решив, что через границы модуля можно транспортировать константы, переменные, процедуры и типы, нужно далее заняться вопросом, какая информация должна быть приведена в спецификации интерфейса для каждого объекта выбранных типов. Здесь руководящим критерием должно быть предоставление минимума информации, согласованной с адекватно определенным интерфейсом модуля. Поэтому достаточно, чтобы константы и переменные специфицировали только имя объекта и его тип. Для процедур достаточно лишь повторить обычный заголовок процедуры. Так как назначение модуля состоит в том, чтобы сделать невидимыми пользователю подробности типа, представляется разумным задавать обычно лишь имя типа. Таким образом, типичная спецификация интерфейса модуля, включающая все четыре вида объектов, может выглядеть так:

```
module пример;
   use const предел : целый;
   define type мой_тип;
         var x, y : мой_тип;
         procedure нуль(inout x: мой_тип);
         function слож(x, y : мой_тип): мтип;
end module пример;
```

К сожалению, спецификации типа не совсем такие простые, как можно было бы ожидать, прочитав только что сказанное. Возвращаясь к понятию абстрактного типа данных, заметим, что модуль, в котором определяется такой тип, имеет целью на самом деле ввести имя типа со свойствами, которые, в общем, эквивалентны свойствам предопределенного типа, такого, как целый или логический. Однако из этого следует, что наряду с операциями для типа нужно также специфицировать и множество литералов. Поэтому, если абстрактный тип является перечислимым типом, литералы перечисления должны быть также заданы в спецификации интерфейса, например

> **module** *цвета*;
>   **define type** *цвет* = (*красный, зеленый, синий*);
>   {операции для цветов}
> **end module** *цвета*;

В случае структурных типов литералы типа обычно будут представлены конструкторами так, как это показано в гл. 3. Ясно, что такие конструкторы нельзя дать для объектов абстрактного типа, так как из их использования автоматически следовало бы знание внутренней структуры типа. Присваивание значений абстрактному структурному типу должно осуществляться с помощью обращения к процедуре. Например, рассмотрим модуль для реализации комплексных чисел:

> **module** *комплексные_числа*;
> **define type** *компл*;
>   **function** *нач*(*r, i* : *плав*) : *компл*;
>   {другие операции для *компл*}
> **end module** *комплексные_числа*;

где функция *нач* предназначена для присваивания значений объекту типа *компл,* например *с*:=*нач* (0.0,-3.0). Заметим, что функцией *нач* подразумевается, что пользователь видит комплексные, числа в декартовой форме, но это не обязательно означает, что внутреннее представление тоже декартово. Оно вполне может оказаться полярным.

Последняя оставшаяся трудность, с которой мы сталкиваемся при реализации абстрактных типов, состоит в работе с описаниями подтипа структурного типа. Например, в гл. 3 (разд. 3.3.2) был введен вариантный комбинированный тип *транспорт* для представления информации об автомобилях и фургонах. Объект подтипа, в котором требовался только ва-

риант автомобиль, описывался с помощью ограничения на описание, т. е.

var *мой_авт*: *транспорт*(*авт*);

Если бы *транспорт* был абстрактным типом, экспортируемым из модуля, то такое описание было бы невозможно. Представляется, однако, полезным разрешить этот частный случай в ситуации, когда комбинированный объект должен быть распределен динамически. Поэтому нужно еще несколько расширить спецификацию интерфейса для типа. Возможна следующая синтаксическая форма:

define type *транспорт* = **record**(*вид_транспорта*);

в которой утверждается, что у типа *транспорт* есть варианты, выбираемые с помощью значений перечислимого типа *вид_транспорта*. Разумеется, если *вид_транспорта* определен в том же самом модуле, то он тоже должен экспортироваться. Аналогичное расширение можно предположить для регулярных подтипов, например,

define type *элем_век* = **array**[*целый*];

будет экспортировать регулярный тип *элем_век* с индексным типом *целый* (см. разд. 3.2.2).

5.3.3. Правила именной области действия

Модуль имеет точно такой же статус, как и все программные единицы, которые мы обсудили в предыдущей главе. Поэтому модули могут вкладываться в другие модули, процедуры, функции или блоки. Однако модуль не подчиняется классическим правилам именной области действия, так как он отличается от остальных перечисленных единиц в одном существенном отношении.

При внесении процедуры в программную единицу вводится новый лексический уровень, такой, что все имена во внутренней процедуре безусловно недоступны из внешней единицы. Но это не верно для вложенного модуля, так как модуль создает новую (замкнутую) именную область действия без введения нового лексического уровня. Другими словами, недоступность объектов, локальных для процедуры, является прямым следствием базового механизма процедуры, т. е. эти объекты существуют только тогда, когда процедура фактически активна. Недоступность же объектов, локальных для модуля, обеспечивается только компилятором. По этой причине имя, локальное для внутреннего модуля, может быть сделано доступным

в объемлющей единице простым распоряжением компилятору сделать его видимым, т. е. указанием его в списке **devine**. Аналогично, имя в объемлющей единице может быть сделано видимым во внутреннем модуле указанием его в списке **use.**

Правила именной области действия для модуля иллюстрируются на следующем примере, где имена, доступные в различных точках, указываются в комментариях:

```
module M1;
   define procedure a;
private
   {описать a}
   var b : целый;
   module M2;
      define procedure c;
      use var b : целый;  private
      {описать c}
      {b, c здесь доступны}
    end module M2;
    procedure d;
       var e: целый
       begin
         {a, b, c, d, e здесь доступны}
       end;
     {a, b, c, d здесь доступны}
 end module M1
   {a здесь доступна}
```

Отметим различие между модулем *M*2 и процедурой *d.* Первый должен явно импортировать переменную *b,* в то время как вторая имеет доступ к ней по умолчанию (для процедур предполагаем соблюдение классического правила именной области действия).

Так как модуль не генерирует новый лексический уровень, имя импортируемого объекта нельзя спрятать в модуле, использовав это имя еще один раз для локального объекта. Поэтому следующее описание модуля незаконно:

```
procedure p1;
   var x :  целый;
   module внутренний;
      use x: целый;
   private
      const x=1;
      .
      .
      .
```

```
    end module внутренний;
begin
    …
end
```

так как имя *x* используется дважды в разных смыслах в модуле *внутренний.*

Любой импортируемый в модуль объект должен быть доступен в объемлющей области. Из этого следует, что при передаче объекта внутрь многократно вложенных модулей имя объекта должно быть указано в списке **use** каждого модуля, например:

```
type t =...;
module M1;
    use type t;
private
    module M2;
      use type t;
    private
        module M3;
          use type t
        private
            {тип t здесь доступен)
        end module M3;
    end module M2;
end module M1;
```

И наконец, заметим, что идентификатор каждого типа, который появляется в списке **define** как обозначение типа переменной, константы или функции или в списке параметров процедуры или функции, должен быть определен либо во внешней среде модуля (и в этом случае он должен явно импортироваться списком модуля **use**), либо в самом модуле (и в этом случае он должен также явно появиться в том же самом списке **define**). Аналогично, имя каждого типа, которое появляется в списке **use** как обозначение типа переменной, константы или функции или в списке параметров, должно быть определено во внешней среде модуля и должно также явно импортироваться списком модуля **use**. Эти правила несколько «неуклюжи», и, возможно, кто-нибудь станет доказывать, что компилятор должен разрешать импортировать или экспортировать такие типы неявно. На практике, впрочем, явное указание имен всех объектов в спецификации интерфейса увеличивает ясность программы и помогает избежать случайных ошибок,

5.4. АСПЕКТЫ РЕАЛИЗАЦИИ

Как указывалось ранее, конструкция модуля не вызывает увеличения времени работы готовой программы, так как она реализуется целиком во время компиляции. Увеличение сложности компилятора, реализующего модульный механизм, значительно, хотя и не чрезмерно, Больше всего усложняется работа с лексической таблицей.

Будет полезно рассмотреть неформально, как компилятор смог бы обрабатывать модульную конструкцию. При входе в модуль компилятор маскирует все имена, которые находятся на данный момент в лексической таблице, за исключением имен, перечисленных в списке use. Затем обрабатывается тело модуля таким же образом, как и для других программных единиц, за исключением того, что при выходе из модуля из лексической таблицы удаляются все модульные имена, кроме имен, перечисленных в списке **define.** Описания из модуля обрабатываются так, как если бы они были написаны в описательной части объемлющей программной единицы. Операторное тело модуля помещается впереди операторного тела объемлющей единицы. Поэтому, например, приведенная в разд. 5.3.1 процедура *использование-стека* стала бы компилироваться так, как если бы она была написана без модульной структуры:

```
procedure использование_стека;
   type элем = целый range - 2048..2047;
        идет = целый range 0..100;
   var s : array[1..100] of элем;
       sp : идет;
       x : элем;
   procedure втолкнуть(in x : элем); ...
   procedure вытолкнуть (out x : элем); ...
   begin
     sp = 0;
       .
       .
       .
     втолкнуть(1); вытолкнуть(x); и т. д.
       .
       .
       .
     end;
```

Интересно отметить, что механизм модуля можно легко добавить к языку без этой возможности с помощью препроцессора (Янг, 1981). После проверки спецификаций интерфейса модуля препроцессор удаляет модуль текстуальным переупорядочением модуля и описаний из объемлющей единицы так, как это показано на приведенном выше примере *использование_стека.*

Замкнутая именная область действия модуля моделируется с помощью систематического переименования всех имен из модуля.

5.5. МЕХАНИЗМЫ АБСТРАКЦИИ В ПРАКТИЧЕСКИХ ЯЗЫКАХ

Конструкция классов языка Симула (Дал и др., 1968) является, по-видимому, самым ранним примером механизма абстракции в практическом языке. Класс - это спецификация типа данных, множество операций и некоторые команды инициализации. При описании переменной с помощью типа класс она автоматически инициализируется командами инициализации класса, а определенные для этого объекта операции обозначаются именем объекта, за которым следует имя операции. Это следует сравнить с использованием модуля для реализации абстрактных типов данных, где инициализация выполняется явно вызовом процедуры инициализации (ср. с *восстан* в модуле *тип_время* - разд. 5.1), а имя каждого абстрактного объекта должно передаваться операции модуля как параметр процедуры. Класс, таким образом, является более чистой реализацией абстрактного типа, чем то, чего можно добиться с помощью модуля, но он является менее общим и несколько увеличивает время работы программы. По существу, класс обеспечивает неявное дублирование объектов, предоставляя (по крайней мере в принципе) новое множество операций для каждого объекта, генерируемого классом. Модуль нельзя дублировать, и поэтому операции модуля одни и те же для всех генерируемых объектов. При сравнении модулей с классами доводы в пользу классов базируются, как правило, на эстетических соображениях. Практически модуль полностью отвечает требованиям реализации абстрактных типов данных и, разумеется, имеет по сравнению с классом много других применений. Вслед за Симулой классы были включены в Параллельный Паскаль (Бринч Хансен, 1975а) и в усовершенствованной форме в некоторые недавние экспериментальные языки, такие, как CLU (Лисков и др., 1977) и Alphard (Шоу, Вульф и Лондон, 1977). Последние два языка были разработаны для облегчения проверки правильности программы, в их проектах использованы некоторые идеи аппликативных языков. К области реального времени они, вообще говоря, не имеют отношения.

Идея модуля как упрощение и обобщение класса впервые появилась в языке Модула (Вирт, 1977 и гл. 12). Модуль в этом языке очень похож на модуль, рассмотренный в этой главе, однако его спецификации интерфейса очень рудиментарны, они представлены списком имен, который не может быть отделен от приватной реализационной части. Язык Паскаль Плюс

(Уэлш и Бастард, 1979) интересен особенно тем, что в нем расширенная конструкция модуля разрешает специфицировать наряду с командами инициализации и команды завершения. Кроме того, язык Паскаль Плюс позволяет производить при необходимости копирование как статических модулей, так и динамических классов.

В конечном итоге все «особо надежные» языки обеспечивали различные формы модульных конструкций. Результирующий язык Ада предоставляет вполне стандартный механизм модуля с раздельными спецификациями интерфейса и реализации (см. гл. 13). Основное достижение языка Ада в этой области состоит в разработке крайне сложной возможности раздельной компиляции.

Глава 6.
ПАРАЛЛЕЛЬНОСТЬ

6.1. ПАРАЛЛЕЛЬНОСТЬ В СИСТЕМАХ РЕАЛЬНОГО ВРЕМЕНИ

Параллельная система - это система, в которой считается, что несколько задач или процессов одновременно являются активными. По существу, все системы реального времени параллельны по своей природе. На самом нижнем уровне базовое машинное оборудование отражает параллельность в механизмах систем прерывания. Типичные операции ввода-вывода включают процедуру инициации, за которой следуют завершения (см. гл. 7). В промежуточное время между этими процедурами устройства ввода-вывода работают параллельно с центральным процессором, синхронизация обеспечивается системой прерывания. Представление этой деятельности ввода-вывода на уровне программного обеспечения упрощается, если рассматривать устройство ввода-вывода и главную программу как отдельные параллельные процессы. И конечно, нужно учитывать современную тенденцию предпочитать микропроцессорные конфигурации одному большому мини-компьютеру. Здесь также деятельность на уровне машинного оборудования лучше всего описывать на уровне программного обеспечения в нотации для представления параллельных процессов.

На более высоких уровнях разработки системы мультипроцессорная организация программного обеспечения очень важна для описания систем реального времени, в которых можно идентифицировать ряд отдельных задач. Например, управляющей процессом вычислительной машине может потребоваться выполнять различные задачи, такие, как прямое цифровое

управление (ПЦУ), аварийный просмотр, выдача данных, оптимизация, прокрутка фоновой программы и т. д. Чтобы описать такие задачи на языке программного обеспечения, требуется система обозначений для описания параллельных процессов.

И наконец, последние работы по методологии разработки программ указывают на то, что множество отдельных задач удобнее специфицировать и разрабатывать как набор простых взаимодействующих процессов (см., например, Коулман и др., 1979). Для применения такой методологии важно иметь языковые конструкции, позволяющие описывать большие совокупности процессов.

В данной главе будут обсуждаться языковые конструкции для описания таких взаимодействующих процессов. Мы будем ориентироваться только на самые общие моменты мультипрограммирования, т. е. на спецификацию, создание, завершение и взаимосвязь процессов. Специфическое применение мультипрограммирования к программированию устройств низкого уровня обсуждается в гл. 7, а проблемы, которые встречаются при обработке ошибок в мультипроцессорной системе, рассмотрены в гл. 8. Прежде чем описывать современный подход к организации параллельности с помощью конструкций параллельного программирования, очень полезно рассмотреть, как многозадачные системы программируются в старых языках реального времени, которые этих конструкций не имеют.

6.2. ТРАДИЦИОННЫЙ ПОДХОД К МУЛЬТИПРОГРАММИРОВАНИЮ

Реализация многозадачной системы с использованием последовательного языка высокого уровня, не обладающего явными конструкциями мультипрограммирования, требует использования многозадачной операционной системы (МЗОС). МЗОС обеспечивает способ связывания вместе ряда отдельных последовательных программ в процессе формирования многозадачной системы. Каждая задача в системе описывается отдельной программой. МЗОС устанавливает переключаемую связь имеющегося процессора (или процессоров) с каждой из задач, реализуя требуемое параллельное выполнение. Связь между задачами обычно осуществляется с помощью общей области памяти, куда пишутся и откуда считываются данные. МЗОС предоставляет системные процедуры, которые дают возможность осуществлять доступ к таким областям взаимно исключающим образом. Кроме этого, процессы часто нуждаются в синхронизации; МЗОС предоставляет системные процедуры, которые позволяют их синхронизировать.

Чтобы обсуждать эти идеи на конкретной основе и чтобы ярче показать проблемы, связанные с программированием

параллельных систем, рассмотрим в качестве примера ситуацию, которая часто встречается в обыкновенных системах реального времени. Предположим, что в системе имеется процесс, который генерирует поток данных. Этот процесс, который мы будем называть поставщиком, должен передать данные второму процессу, который будет называться потребителем. Предполагается, что скорость генерации и потребления данных подвержена флуктуациям, поэтому для сглаживания этих флуктуации между обоими процессами помещается ограниченный буфер. Реализация такого буфера нас здесь не интересует. Будем просто считать, что элемент данных x заносится в буфер обращением *занести* (x) и элемент данных выбирается из буфера с помощью *выбрать* (x). Так как буфер ограничен, необходимы еще две функции: *полон,* значение которой истинно, когда буфер полон, и *пуст,* значение которой истинно, когда буфер пуст.

Первое решение данной задачи о поставщике/потребителе можно в общих чертах описать, как показано ниже. Здесь каждый процесс представлен отдельной программой, и предполагается, что буфер постоянно находится в области общей памяти.

```
{задача поставщик/потребитель — решение 1}
common  var b: буфер;
end
program поставщик;
   var x : данные;
begin
  do
     выработать(x);
     while полон do
           ждать
       end do;
    занести(x)
  end do
end
program потребитель;
   var x : данные;
begin
  do
     while пуст do
           ждать
       end do;
    выбрать(x);
    потребить(x)
  end do
end
```

Процесс поставщик выполняет бесконечный цикл: выработка данного *x*, ожидание состояния неполного буфера, занесение *x* в буфер. Процесс потребитель выполняет соответствующий дополнительный цикл: ожидание непустого буфера, выбор из буфера данного *x* и его обработка.

Конечно, это решение не работоспособно, так как не гарантирована невозможность одновременного доступа к буферу, что может привести к неправильному выбору или занесению данных. Традиционное решение того, как обеспечить взаимно исключающий доступ к общим ресурсам, таким, как общий буфер, состоит в использовании семафоров (Дейкстра, 1968). Семафор - это двоичная переменная, которая регистрирует, является или нет доступным ресурс, который она защищает. Для заданного семафора *s* ресурс доступен, когда $s = 1$; в противном случае он недоступен, и процесс, который намерен к нему обратиться, должен ждать. Чтобы процесс не тратил напрасно процессорное время, ожидание организуется следующим образом: выполнение процесса приостанавливается и он помещается в очередь, связанную с этим семафором. Проверка процессом семафора *s* выполняется с помощью обращений к двум системным процедурам:

оградить(*s*) **if** *s* = 1 **then** s := 0 **else**
приостановить вызывающий процесс и
поместить его в очередь(*s*)

освободить(*s*) **if** *очередь*(*s*) *пуста* **then** *s*:= 1 **else**
возобновить первый процесс в очереди(*s*)

Поддерживая целостность семафора, операционная система выполняет эти процедуры взаимно исключающим образом.

С использованием семафора взаимно исключающий доступ к буферу для нашего примера может быть обеспечен путем заключения обращений к процедурам доступа к буферу между обращениями к процедурам *оградить* и *освободить*. Но, прежде чем переписать решение 1 так, чтобы воспользоваться этими процедурами, нужно рассмотреть еще один вопрос. В решении 1 ситуации, когда буфер пуст и полон, разрешаются с помощью «рабочего ожидания». Ясно, что это очень плохо, так как напрасно тратится процессорное время. Подобного можно избежать, если воспользоваться еще одним видом примитива, называемого сигналом. Сигналы используются процессом для сообщения другому процессу факта совершения некоторого события. Для работы с сигналами имеются две операции:

послать(*e*) - *послать сигнал e*
ждать(*e*) - *ждать поступления сигнала e*

Впрочем, точной семантики сигналов нет; следующие определения характерны для тех трактовок, где предполагается, что сигнал *e* есть двоичная переменная, принимающая значение 1, когда сигнал *e* был послан, но еще не получен:

```
послать(e): if очередь(e) пуста then e:=1 else
            возобновить первый процесс в очереди(e)
ждать(e):   if e = 1 then e := 0 else
            приостановить вызывающий процесс и
            поместить его в очередь(e)
```

Очевидно, что определение сигнала очень похоже на определение двоичного семафора. Различие заключается не в том, как он реализуется, а в том, как он используется. Семафор используется для реализации взаимного исключения, и один и тот же процесс будет обращаться к процедурам *оградить* и *освободить*. Сигнал используется для синхронизации двух процессов. Один процесс обратится к процедуре *послать,* а второй, чтобы с ним синхронизироваться, обратится к процедуре *ждать*.

С использованием этих примитивов семафор и сигнал второе (и на этот раз правильное) решение задачи о поставщике/потребителе может быть сформулировано следующим образом:

```
{задача поставщик/потребитель — решение 2}

 common
    var b : буфер;
        s : семафор;
        неполон, непуст: сигнал;
end
 program поставщик;
    var x : данные;
 begin
    do
      выработать(x);
      оградить(s);
      if полон then
         освободить(s);
        ждать (неполон);
        оградить(s)
      end if;
      занести(x);
      освободить(s);
      послать (непуст)
    end do
  end
```

```
program потребитель;
   var x: данные;
begin
  do
    оградить(s);
     if пуст then
       освободить(s);
       ждать(непуст)
       оградить(s)
     end if;
     выбрать(x);
    освободить(s);
    послать(неполон)
  end do
end
```

В этом решении процесс поставщик в нормальной ситуации выполняет последовательность

выработать(*x*);
оградить(*s*);
проверить, не полон ли буфер;
занести(*x*);
освободить(*s*)

где так называемый критический код заключен между обращениями к процедурам *оградить* и *освободить,* что обеспечивает исполнение этого кода без взаимодействия с другим процессом. Усложнение программы происходит главным образом из-за необходимости синхронизировать поставщик для случая, когда буфер становится полным. В этой ситуации поставщик станет ждать сигнала *неполон.* Заметим, что прежде чем перейти в состояние ожидания, он должен освободить семафор, иначе в системе возникнет тупиковая ситуация, так как потребитель может освободить поставщика только считыванием с буфера элемента данных. Это было бы невозможно, если бы поставщик по-прежнему держал семафор закрытым. В конце каждого цикла потребителя посылается сигнал *неполон.* В обычной ситуации этот сигнал ни на что не повлияет; но в том специальном случае, когда поставщик ждет свободного места в буфере, он получит этот сигнал и продолжит свою работу. Выполнение процесса потребителя в точности соответствует работе поставщика с той поправкой, что он должен синхронизироваться только, когда буфер пуст.

Это решение задачи поставщик/потребитель типично для методологии, которая должна использоваться, когда требования

взаимного исключения и синхронизации удовлетворяются с помощью таких примитивов операционной системы, как семафоры и сигналы. Исследование недостатков этого подхода позволяет лучше понять причины поиска альтернативных решений, описываемых в следующих разделах.

Основной недостаток этого решения — неясность его функционирования, что при его разработке может легко привести к появлению ошибок. Это объясняется очень низким уровнем примитивов семафор и сигнал. Рассмотрим некоторые из ошибок, которые легко сделать, разрабатывая программы с помощью семафоров:

(1) Можно передать управление в критический участок и, следовательно, обойти необходимое обращение к процедуре *оградить.* В результате к общим данным может быть осуществлен доступ одновременно двумя процессами, что приведет к порче данных.

(2) Можно забыть обратиться к процедуре *освободить,* что приведет к тупиковой ситуации.

(3) В сложных программах можно забыть воспользоваться семафором.

Аналогичные ошибки можно также сделать при использовании сигналов. Более того, кроме присущей им ненадежности и нечеткости использования, они имеют еще два существенных недостатка. Во-первых, нельзя программировать альтернативное действие для случая, когда семафор окажется занятым, и, во-вторых, нельзя ждать, пока один из семафоров не окажется свободным.

Итак, семафоры и сигналы являются примитивами низкого уровня; их можно легко использовать неправильно, но даже и при правильном использовании их применение ведет к получению неясных программ. Более того, механизм семафоров недостаточен для описания некоторого вида программных действий.

Кроме только что рассмотренных проблем решение 2 демонстрирует и несколько более общих недостатков данного подхода. Во-первых, ответственность за правильный доступ к общему буферу лежит на процессе, который осуществляет этот доступ. Ясно, что это противоречит практике программирования, описанной в гл. 5, где защищалась та точка зрения, что данные и все операции над ними должны объединяться в одну единицу. Во-вторых, следует еще раз подчеркнуть, что каждая программа компилируется отдельно, причем компилятору ничего не известно о параллельной среде. Поэтому у компилятора нет никакой возможности проверить правильность использования «параллельных» примитивов. Другим следствием принудительной раздельной компиляции является то, что разработчик программной

системы будет пытаться минимизировать общее число различных процессов в системе, так чтобы количество внешних связей и программ интерфейса с МЗОС не превышало возможности управления. Это часто будет идти вразрез с естественным разбиением задачи, которая, возможно, лучше бы описывалась с помощью значительно большего числа мелких процессов.

Из нашего рассмотрения с очевидностью следует, что добиться прогресса, предоставив программисту для описания параллельных систем более совершенные средства, можно только в том случае, если ввести в сам язык программирования конструкции параллельного программирования. Эти конструкции позволят представлять несколько процессов в одной программе и дадут простые и надежные механизмы общения между процессами. Кроме того, поскольку компилятор тогда будет располагать сведениями об особенностях параллельной среды, появляется возможность проверки правильности использования этих конструкций во время компиляции. В следующем разделе обсуждаются механизмы описания параллельных процессов в рамках одной программы. В дальнейших разделах разбирается общение между процессами.

6.3. ПРОЦЕССЫ

На концептуальном уровне процесс можно представлять как аналог процедуры. Описание процедуры определяет множество действий; отдельное понятие вызова процедуры есть возбуждение этих действий. При применении аналогичного подхода к процессам мы получим описание процесса, определяющее множество последовательных действий, и вызов процесса, который возбуждает эти действия. Рассмотрим, например, следующий эскиз программы, в которой процесс с именем *A* описывается, а затем возбуждается в основном теле программы обращением к нему:

```
program пример_процесса_1;
  process A;
  begin
     …
  end;
  begin
    A;
    …
  end
```

Особенность описания *A* как процесса, а не как процедуры состоит в том, что при вызове *A* порождается новый «механизм управления», с помощью которого вызванный процесс

выполняется параллельно с вызывающей программой. При вызове процедуры это не так; в случае процедуры выполнение вызывающей программы приостанавливается до тех пор, пока не завершится выполнение вызванной процедуры

Проводя чуть дальше аналогию с процедурой, заметим, что процесс можно специфицировать с помощью параметров и повторными обращениями можно реализовать различные возбуждения процесса. Например,

```
program пример_процесса_2;
   type ид_задачи = (просмотр1, просмотр2, просмотр3);
   process задача(in ид : ид_задачи);
   begin
      …
   end;
begin
   задача(просмотр1); задача(просмотр2);
   задача(просмотр3);
   …
end
```

является программой, в которой процесс с именем *задача* вызывается три раза, каждый раз с другим входным параметром. Поэтому здесь порождаются три новых механизма управления, что дает в результате четыре параллельных механизма управления. Заметим, однако, что параметры процесса по необходимости ограничиваются параметрами класса **in.** Вызывающему процессу значения возвращаться не могут, так как он не дожидается, пока они поступят.

В рассмотренных двух примерах главная программа называется родительским процессом, а процессы, которые она вызывает, называются порожденными процессами. Часто бывает удобно при необходимости определить иерархию процессов, где каждый процесс определяется в терминах следующих порожденных процессов. В этих случаях требуются правила, относящиеся к времени жизни процесса

Процесс, как правило, завершается, когда управление достигает его конца. Если процесс создал порожденные процессы, то он завершается лишь тогда, когда достигается его собственный конец и когда завершаются также все порожденные им процессы. Это правило гарантирует, что при завершении порожденного процесса родительский процесс может считать, что и все дальнейшие процессы (т. е. «внуки»), порожденные завершенным процессом, тоже завершены.

Наше расширение понятия процедуры, охватывающее представление параллельных процессов, имеет то достоинство, что оно относительно просто. И в самом деле, подобные механизмы

с успехом использовались в различных языках. Но, к сожалению, в контексте языков реального времени это расширение имеет два существенных дефекта. Во-первых, процессы, создаваемые при повторных обращениях к процессу, являются, по существу, анонимными: у них нет имен, по которым другие процессы могли бы на них ссылаться. Это не является проблемой в нормальных обстоятельствах, где два взаимодействующих процесса могут установить информационную связь между собой и через этот канал влиять на работу друг друга. Но в системах реального времени должен иметься специальный механизм, с помощью которого один процесс может влиять на другой процесс независимо от того, согласен второй процесс с первым взаимодействовать или нет. Например, весьма необходима некоторая форма механизма прекращения, чтобы процесс *A* мог задать обращение *прекратить* (*B*) для завершения некоторого «разбойничьего» процесса *B*. Ясно, что для того, чтобы этот механизм смог работать, все процессы должны обладать некоторой уникальной формой идентификации. Во-вторых, разрешение создавать любое количество экземпляров процесса вызывает реализационные трудности, особенно когда любой процесс может создавать любое число порожденных процессов.

При создании процесса требуется место в памяти для персонального стека. Это место не может быть выбрано в стеке, принадлежащем родительскому процессу, так как общий объем памяти, которая может потребоваться, нельзя предсказать заранее. Поэтому для поддерживания создания процессов по этой схеме необходимо управление динамической памятью.

Очевидное решение двух этих проблем состоит в разрешении только одной фактической активации процесса в каждый момент времени [4]. На этот процесс, таким образом, можно сослаться с помощью имени, находящегося в описании процесса. Однако уже само по себе это решение необоснованно узко, так как ясно, что имеются области применения, где требуется создавать несколько одинаковых процессов. Удобным расширением, которое позволяет это сделать, является понятие массива процессов. Например, программу *пример_процесса_*2 можно переписать следующим образом:

```
program пример_процесса_3;
   type ид_задачи = (просмотр1, просмотр2, просмотр3);
   process задача[ид_задачи];
   begin
   …
   end;

begin
   init задача[просмотр1], задача[просмотр2],
        задача[просмотр3];
   …
end
```

[4] При этом одновременно могут выполняться только процессы с разными именами. — *Прим. ред.*

где из описания процесса *задача* удалены формальные параметры, так как они больше не нужны, и вместо этого добавлен дискретный диапазон. Поэтому описание *задача* указывает, что может быть создано самое большее три экземпляра процесса. На каждый экземпляр ссылаются с помощью имени *задача,* за которым следует индекс в квадратных скобках. В результате на каждый экземпляр процесса можно сослаться по уникальному имени, а требования к объему общей стековой памяти могут быть определены заранее, что облегчает реализацию.

В этом примере имеется еще одно изменение — создание процесса обозначается явно оператором **init.** Причины его введения не столь очевидны. Если бы процесс каждой задачи инициировался одним вызовом процесса, то тогда обязательно было бы так, что один процесс стал бы активным прежде, чем два других. Если бы этот процесс попытался установить связь с одним из них прежде, чем они были созданы, могла бы возникнуть ситуация с ошибочной операцией. Этого можно избежать, если ввести оператор, который позволяет осуществлять одновременную активацию нескольких процессов.

Описанный механизм соответствует задачам, возникающим в основных областях реального времени; он достаточно гибок и способствует относительно эффективной реализации. Можно привести примеры других механизмов, с другим соотношением между гибкостью и эффективностью. Например, широко распространенным методом является разрешение создавать процессы точно таким же способом, как и динамические объекты данных, ссылаясь затем на них с помощью указательных переменных. Впрочем, здесь мы этим заниматься не будем. Более значительной темой является общение процессов, этот вопрос мы и будем рассматривать следующим.

6.4. МОНИТОРЫ

В разд. 6.2 было описано использование примитивов низкого уровня для программирования общения процессов и обсуждены их недостатки. Этими примитивами были семафоры для программирования взаимно исключающего доступа и сигналы для программирования синхронизации. Из этих двух примитивов наибольшие трудности на практике создают семафоры. Поэто-

му для решения вопросов взаимного исключения были предложены различные конструкции высокого уровня. Ранние предложения концентрировались вокруг семафорного принципа, вводя синтаксические ограничения на его использование. Например, было предложено понятие критической области, где операции *оградить* и *освободить* неявно играли роль скобок для последовательности операторов. Это исключает ошибочный обход операции *оградить* и не позволяет случайно опустить операцию *освободить,* но при решении многих практических задач, связанных с параллельностью, критические области оказались недостаточно гибкими.

В гл. 5 были введены понятия механизмов абстракции, с помощью которых данные и множество операций, которые могут применяться к этим данным, объединяются в одно целое. Целостность данных можно гарантировать лишь тогда, когда доступ к данным разрешается только на основе разрешенного множества операций. Эта идея была перенесена на область параллельных систем введением понятия монитора, предложенного впервые Бринчем Хансеном (1973, 1975а) и Хоаром (1974). Монитор — это модуль, в котором общие данные объединены вместе с множеством процедур, реализующих доступ к этим данным. Процедуры монитора обладают специальным свойством: каждый раз только один процесс может активно выполнять мониторную процедуру в данном мониторе. Следовательно, мониторы являются конструкцией высокого уровня, позволяющей осуществлять взаимно исключающий доступ к разделяемым ресурсам. Однако они не предоставляют никаких средств синхронизации процессов, и поэтому мониторная конструкция должна использоваться вместе со средством, позволяющим обрабатывать в мониторе сигналы.

Использование мониторов лучше всего показать на примере. Следующая программа является решением введенной в разд. 6.2 задачи поставщик/потребитель; для краткости в ней опущены некоторые подробности, такие, как спецификация интерфейса монитора (ср. со спецификациями интерфейса модуля — гл. 5).

```
program пост_потр; {решение_3}
  monitor буферизация;
    var b: буфер;
          неполон, непуст: сигнал;
procedure entry передать (in x: данные);
  begin
    if полон then ждать(неполон) end if;
    занести(x);
    послать(непуст)
  end;

  procedure entry получить(out x : данные);
  begin
    if пуст then ждать(непуст)  end if;
    выбрать(x);
    послать(неполон)
  end;
  begin
     инициализировать b
  end; {монитора}

  process поставщик
    var x: данные;
  begin
    do
      выработать(x);
      буферизация.передать(x)
    end do
  end;
```

```
        process потребитель
          var x : данные;
        begin
          do
            буферизация.получить(x); потребить(x)
          end do
        end;
    begin
    init поставщик, потребитель
  end
```

В этом решении процессы поставщик и потребитель описаны и активизированы при помощи механизма, рассмотренного в предыдущем разделе. Структура этих процессов теперь значительно проще, чем в прежних решениях; каждый процесс состоит только из двух обращений к процедурам. Ответственность за все обращения к общему буферу полностью берет на себя монитор.

Сам монитор построен как модуль. В модуле содержатся описания данных, входные процедуры для обеспечения доступа к данным и основное тело, которое выполняется, когда описание модуля используется для инициализации данных. Опишем работу модуля, не обращая внимания пока на ситуацию, когда буфер пуст или полон. Процесс поставщик передает данные в буфер, обращаясь к мониторной процедуре *передать.* Аналогично, процесс потребитель передает данные с буфера, обращаясь к мониторной процедуре *получить.* Если какой-нибудь из этих процессов попытается обратиться к любой из названных процедур, когда другой процесс находится уже в мониторе, выполняя мониторную процедуру, то его выполнение

будет приостановлено и он будет помещен в очередь. Каждый раз, когда процесс покидает монитор, происходит проверка очереди; если там имеется ждущий свободного монитора процесс, то этот процесс активизируется. Таким образом осуществляется взаимно исключающий доступ к общим данным процессов.

Чтобы иметь возможность осуществлять в мониторе синхронизацию, вводится механизм сигнализации. Когда процесс получает доступ к процедуре монитора и затем задает операцию ждать сигнала, то правило взаимного исключения игнорируется и следующему процессу разрешается войти в монитор. Выполнение ожидающего процесса может быть возобновлено лишь при вхождении другого процесса в монитор и посылке сигнала, что он ждет. Когда это происходит, в мониторе становятся активными два процесса; поэтому для поддержания требуемого взаимного исключения вводится правило, согласно которому последним выполнимым оператором мониторной процедуры должна быть операция *послать.*

В рассматриваемом примере программы обращение к процедуре *передать* приведет к приостановке вызывающего процесса до тех пор, пока не будет прислан сигнал *неполон,* в случае переполненного буфера. Этот сигнал посылается каждый раз при выполнении процесса *получить,* чтобы выполнение каждого ждущего процесса могло начаться как можно быстрее. Если ждущих процессов нет, то наличие этого сигнала ни на что не влияет. Аналогичный механизм используется для обработки соответствующего условия для пустого буфера.

Задача поставщик/потребитель является типичным примером того, как используются мониторы. Решение практических задач подтвердило, что с помощью мониторов можно удовлетворительно решать самые разнообразные проблемы; в проектах современных параллельных языков программирования мониторы являются, по-видимому, наиболее широко используемым механизмом взаимного исключения. Причина популярности мониторов связана с рядом их удачных особенностей. Эти особенности проистекают главным образом из модульной структуры монитора. Так как все обращения к общим данным должны выполняться с помощью мониторных процедур, становится возможным написать и отладить монитор отдельно от процессов, которые им будут пользоваться. При введении монитора в систему гарантируется целостность данных, которые он защищает, а правильность его функционирования не может быть нарушена при добавлении к системе содержащего ошибки процесса. Компилятор может проверить, чтобы все обращения к общим данным были законными; более того, он может построить граф доступов к системе и, по крайней мере в принципе, обнаружить возможные тупиковые ситуации. Конечно, понятие монитора

по-прежнему предполагает использование сигналов, но они теперь хотя бы заключены в рамки монитора и тем самым эффективно удалены из главной части системы.

Несмотря на перечисленные достоинства, понятие монитора ставит две очень трудные проблемы. Во-первых, выходящий из монитора процесс может возобновить самое большее один другой процесс. Соблюдение этого правила необходимо для сохранения свойства взаимной исключаемости, но в некоторых областях применения ведет к различным трудностям. Примером этого является ситуация, когда необходимо синхронизировать несколько процессов с помощью процесса, управляющего синхронизацией. Эту проблему можно обойти, если допустить более широкое использование сигналов, т. е. не ограничивать их использование только рамками монитора; но тогда снова возникают трудности, описанные в разд. 6.2.

Во-вторых, общая методология введения в язык механизма абстракции, такого, как монитор, подразумевает, что программы разрабатываются методом сверху вниз, образуя иерархию абстрактных типов данных. Если мы хотим, чтобы монитор удовлетворял этой схеме, то должна иметься возможность вызывать из монитора *А* содержащуюся в мониторе *В* процедуру. И здесь мы сталкиваемся с одним хорошо известным недостатком мониторов, известным под названием *проблемы вложенных мониторов.* Если вызывающий процесс в *В* приостановлен, то, хотя взаимное исключение освобождает процесс в *В,* оно не освобождает процесс в *А.* Поэтому, если доступ к монитору *В* возможен только через монитор *А,* в системе неизбежно возникает тупиковая ситуация.

В связи с этими двумя проблемами и нежеланием вводить примитивное понятие сигнала в последнее время получает распространение другой подход к реализации общения между процессами. Этот подход мы рассмотрим в следующем разделе.

И наконец, следует отметить, что при условии запрещения обращений к вложенным мониторам монитор может быть реализован очень эффективно как на одном процессоре, так и на мультипроцессорном машинном оборудовании с общей памятью. Демонстрировалась также возможность реализации мониторов в системах со слабо связанными распределенными процессорами (Даусон и др., 1979), но их использование в таких системах несколько неестественно. Монитор предназначается для защиты общей памяти, а в распределенных системах нет общей памяти, которая нуждалась бы в защите. Так как применение распределенного машинного оборудования становится все более и более популярным, имеется еще одна причина для поиска другого подхода.

6.5. РАНДЕВУ

Во многих отношениях разработка механизмов общения между процессами находилась под влиянием традиционной реализации этих механизмов. Обычным был такой случай, когда мультипроцессорное программное обеспечение всегда реализовалось на машинах с одним процессором, и поэтому передача данных между процессами осуществлялась через области общей памяти. У разработчиков языка имелась тенденция неявно ориентироваться на эту модель, что отражалось в тех конструкциях высокого уровня, которые ими разрабатывались.

В более поздних работах Хоара (1978) и Бринча Хансена (1978) было показано, что можно пойти по более естественному пути при решении проблемы межпроцессной связи, если рассматривать передачу данных и синхронизацию как одну неразделяемую деятельность. Модель, которую они предлагают, требует, чтобы, в случае когда процесс *A* намерен передать данные процессу *B, оба* процесса должны выразить свою готовность установить связь, выдав заявки соответственно на передачу и получение. Если получится так, что процесс *A* выдаст заявку на передачу первым, то его выполнение приостанавливается до тех пор, пока процесс *B* не выдаст заявку на получение. Аналогично, если первым выдаст заявку на получение процесс *B,* то он также приостанавливается до тех пор, пока процесс *A* не выдаст заявку на передачу. Когда оба процесса таким образом синхронизируются, происходит передача данных и каждый из процессов возобновляет свою деятельность. Такую синхронизацию называют рандеву. Фактическая передача данных между процессами происходит без использования буфера, поэтому передача данных между асинхронными процессами должна программироваться с привлечением буферного процесса, работающего между ними. Таким образом, как показано далее, пассивная мониторная конструкция, которую мы использовали в решении задачи поставщика/потребителя в предыдущем разделе, заменяется активным процессом на базе модели рандеву.

Модель рандеву, предложенная Хоаром, реализует взаимодействие процессов симметрично в том смысле, что оба вступающих в общение партнера обрабатываются одинаково.

В качестве примера рассмотрим следующий эскиз программы, в котором процесс *A* намерен передать данное *x* процессу *B* (отметим, что сам Хоар использует другую нотацию, значительно более компактную, чем та, которая применяется в нашем примере):

```
process A;
  var x: данные;
begin
  B!x;
  …
end;

process B;
  var y : данные;
begin
  …
  A?y;
  …
end;
```

Оператор *B*!*x* в процессе *A* означает выдачу значения данных *x* процессу *B,* а соответствующий оператор *A?y* в процессе *B* означает получение некоторого значения данных от процесса *A* и хранение этого значения в *y*. Чтобы передача данного состоялась, процессы должны синхронизироваться посредством рандеву. Каждый процесс, который выдаст команду передачи первым, приостанавливается до тех пор, пока второй процесс не выдаст свою команду соответствующей передачи. Предполагается, что затем фактическая передача данного происходит мгновенно, после чего оба процесса продолжают свою работу. Такой механизм симметричен; чтобы состоялось рандеву, каждый процесс должен назвать имя другого. Для практических языков реального времени это является ограничением и ведет к серьезной проблеме, так как становится невозможным написать общий «библиотечный процесс», который обслуживает процессы пользователя с неизвестными ему именами.

Альтернативой подходу симметричной связи является асимметричная связь, где только один процесс (хозяин) называет имя второго процесса (служащего) при назначении рандеву. Этот подход позволяет писать библиотечные процессы, и его взял за основу Бринч Хансен в проекте системы процессов языка Ада. Мы будем развивать асимметричное рандеву, базируясь на модели из языка Ада. Впрочем, мы будем обсуждать только основные механизмы и примеры будем давать в обычной нейтральной нотации языка Паскаль. Более подробное обсуждение системы процессов языка Ада можно найти в гл. 13.

На уровне языка асимметричное рандеву может быть реализовано с помощью включения в процесс-служащий оператора «приема» [5]. Оператор приема имеет форму входной процедуры, так что процесс-хозяин может общаться с процессом-служащим, обращаясь к одному из его операторов приема. Приве-

---

[5] С использованием служебного слова accept (принять). — *Прим. ред.*

денный выше пример симметричного рандеву в несимметричном случае имел бы следующий вид:

```
process A;
   var x: данные;
begin
   …
   B.послать(x)
   …
end;
process B;
   var y: данные;
begin
   accept послать(in элем : данные);
         y:=элем
end;
   .
   .
   .
end;
```

Внутри процесса-хозяина *A* данное *x* передается с помощью обращения к процедуре точно так же, как и в случае использования монитора. Внутри процесса-служащего *B* должен быть выполнен соответствующий оператор приема, чтобы инициировать рандеву и в результате получить элемент данных. Фактическая передача данных выполняется точно таким же образом, как и при обычном вызове процедуры, т. е. фактические параметры в обращении к процедуре ставятся в соответствие с формальными параметрами из спецификации оператора приема. После передачи данных оба процесса остаются занятыми рандеву, пока исполняется тело оператора приема (концептуально оно исполняется процессом-служащим, хотя на практике это может быть сделано и процессом-хозяином). В этом случае тело оператора приема выполняет работу по копированию локальной переменной в глобальную переменную *y*. После завершения оператора приема оба процесса возобновляют свою самостоятельную деятельность.

В этом описании асимметричного рандеву появились два новых понятия. Во-первых, передача данных специфицируется с помощью параметров процедуры. Можно специфицировать любое число параметров, которые могут принадлежать классу **in, out** или **inout.** При одном рандеву передача данных может идти в обоих направлениях. Во-вторых, тело оператора приема выполняется эффективно как критический участок. Это необходимо, так как параметры оператора приема строго в нем-

локализованы; но это так же и очень полезно, так как обеспечивает дополнительную гибкость программирования. Поэтому такое определение модели рандеву более точно можно назвать определением расширенного рандеву.

Основными преимуществами механизма межпроцессного взаимодействия такого вида считаются сравнительная легкость понимания функционирования сложных параллельных программ и доказательства их правильности, так как при взаимодействии процессы должны синхронизироваться. Поэтому при посылке процессу сообщения программист не может оказаться в состоянии неопределенности относительно состояния этого процесса. Он знает ответ заранее: процесс должен находиться в состоянии выполнения оператора рандеву (вызов или прием), а процесс, который послал сообщение, также должен находиться в состоянии рандеву.

Впрочем, очевидно, что эта очень простая модель оказалась бы недостаточной при программировании реальных задач. Очень строгая синхронизация процессов не дает возможности выполнять различные асинхронные операции и, следовательно, не позволяет реализовать все имеющиеся в программе возможности параллельного исполнения. При решении этой проблемы принцип рандеву сомнению не подвергают, но идут по пути введения возможности недетерминированного выбора операторов приема. Другими словами, вводится механизм, с помощью которого процесс-служащий может не выполнять оператор приема до тех пор, пока про процесс-хозяин не станет известно, что он действительно ждет; при этом самому процессу-служащему уже не нужно ждать процесс-хозяина на рандеву.

Чтобы проиллюстрировать это, рассмотрим систему, в которой к переменной *общая_пер* должен иметься свободный доступ как на чтение, так и на запись любому числу процессов-хозяев; но каждый индивидуальный доступ к этой переменной должен выполняться со взаимным исключением. Одно из решений этой задачи состоит в том, чтобы поместить нашу переменную в процесс, единственной функцией которого является защита этой переменной. Требуемый процесс кодируется следующим образом:

```
process защищенная_пер;
   var общая_пер : данные;
begin
   do
      select
         accept читать (out x : данные);
            x := общая_пер
         end
      or
        accept писать(in x : данные);
                 общаяпер := x
        end
      end select
    end do
  end;
```

Этот процесс удовлетворит запросы на чтение и запись общей переменной в любом порядке. Его функционирование зависит от оператора выбора[6], который был введен для обеспечения недетерминированного выбора операторов приема. Каждый раз при выполнении оператора выбора имеется четыре возможности:

(1) Ожидание обращения на чтение.
(2) Ожидание обращения на запись.
(3) Ожидание обращения как на чтение, так и на запись.
(4) Нет обращений ни на чтение, ни на запись.

В случаях (1) и (2) немедленно выполняется соответствующий оператор приема. В случае (3) выбирается случайным образом один из двух операторов приема, который затем и выполняется. В случае (4) процесс приостанавливается до тех пор, пока не поступит запрос либо на чтение, либо на запись; в момент поступления выполняется соответствующий оператор приема. Следовательно, каждый раз при выполнении оператора выбора будет выполняться в точности одна процедура приема; но выбор самой процедуры и времени ее выполнения зависит от характера последовательности обращений.

Рассмотренный пример иллюстрирует тот случай, когда выполнение оператора приема требует только наличия ожидающего обращения. В некоторых ситуациях может потребоваться учитывать дополнительное условие, характеризующее состояние структуры данных в процессе-служащем. Такие условия реализуются за счет того, что разрешается помещенный в оператор выбора оператор приема снабжать предшествующим ему предохранителем. Состояние предохранителя обозначается значением логического выражения, причем предохранитель открыт в случае истинности выражения и закрыт (не пропускает), когда оно ложно. Отсутствующий предохранитель (как в предыдущем примере) всегда считается открытым. При входе в оператор выбора производится вычисление всех предохранителей, и затем в качестве кандидатов для обработки рассматриваются только те операторы приема, которым предшествуют открытые предохранители.

---

[6] Со служебным словом **select** (выбрать). — *Прим, ред.*

Как было установлено ранее, решение задачи поставщик/потребитель с применением модели рандеву требует, чтобы использованный в еще более раннем решении пассивный монитор был заменен активным процессом, в обязанности которого входило бы управление доступом к буферу. Мониторные процедуры *передать* и *получить* заменяются операторами приема, но, так как обращение к *передать* может быть принято только в том случае, когда буфер неполон, и так как обращение к *получить* может быть принято только в том случае, когда буфер непуст, этим операторам приема должны предшествовать предохранители. Процесс, эквивалентный мониторной буферизации, имеет следующий вид:

```
process буферизация;
   var b: буфер;
begin
   инициализировать b;
   do
      select
         [not полон]
         accept передать(in x ; данныё);
           занести(x)
         end
      or
         [not пуст]
         accept получить(out x : данные);
            выбрать(x)
         end
      end select
   end do
end;
```

причем предохранители написаны непосредственно перед операторами приема в квадратных скобках. Обычно буфер заполнен лишь частично, и, следовательно, оба предохранителя открыты; данный процесс работает как описанный ранее процесс *защищенная_пер*. Однако если буфер пуст, то оператор приема *получить* закрыт предохранителем и процесс будет ожидать до тех пор, пока не произойдет обращение к *передать,* если даже при этом придется ждать обращение к *получить.* Ясно, что если бы были закрыты оба предохранителя, то процесс приостановился бы на неопределенное время; поэтому, как и в нашем случае, логика программы должна быть такова, чтобы этого никогда не случилось.

В нашем описании расширенного асимметричного рандеву появились два важных понятия. Первое состоит в том, что взаимодействие процессов полностью синхронизировано и ини-

циируется одним процессом, который выдает обращение к оператору приема, выполняемому вторым процессом. Второе состоит в существенной необходимости недетерминированного выполнения операторов приема, которое может быть реализовано в операторе выбора. Там, где это удобно, можно воспользоваться дополнительной возможностью управления работой оператора выбора с помощью предохранителей.

Механизм рандеву является, по-видимому, более чистым решением фактически всех проблем мультипрограммирования, чем предыдущие, ориентированные на взаимное исключение механизмы, такие, как семафоры, критические участки и мониторы. Кроме того, механизм рандеву столь же удобен для использования при распределенно-мультипроцессорных, мультипроцессорных с общей памятью и однопроцессорных архитектурах. Его единственным существенным недостатком является то, что решения с использованием этого механизма требуют, как правило, больше параллельных процессов, чем сравнимые решения, в которых используются другие механизмы. В результате могут оказаться очень высокими накладные расходы, связанные с контекстными переключениями физического процессора (или процессоров) между всеми параллельными процессами. Это хорошо видно на примере задачи поставщик/потребитель, где пассивные процедуры доступа к буферу первых решений пришлось заменить на активные процессы в контексте аппарата рандеву. Впрочем, как показано в разд. 6.8, часто бывает возможна оптимизация на уровне компиляции, что делает данный недостаток менее существенным.

6.6. ПРОХОЖДЕНИЕ СООБЩЕНИЙ

Предшествующее обсуждение механизмов программирования взаимодействия процессов было сконцентрировано на конструкциях общего назначения, которые достаточно гибки, чтобы их можно было применять при построении существенно разнообразных многопроцессных систем. Так, например, механизм рандеву может быть использован при реализации синхронизации процессов (непараметризованные операторы приема), передаче сообщений (ср. процесс *буферизация*) и управлении общими ресурсами (ср. процесс *защищенная_пер*).

Для языков, ориентированных на очень узкую область применения, можно пытаться выбрать в качестве средства межпроцессной коммуникации менее общий механизм высокого уровня. Типичным примером таких ориентированных на применение механизмов являются системы прохождения сообщений, в которых все взаимодействия процессов осуществляются с помощью передачи сообщений. Определяя систему прохождения

сообщений в языке высокого уровня, обычно вводят новую конструкцию структурирования данных, называемую *буфером,* которая предназначается для спецификации межпроцессных буферов. Например, описание

var *почтовый_ящик* : **buffer**[10] **of** *сообщение*;

приведет к созданию буфера с именем *почтовый_ящик,* в котором можно хранить до десяти объектов типа *сообщение.* Один процесс может послать другому процессу сообщение с помощью предопределенной процедуры

послать(*x, почтовый_ящик, p*)

где *x* обозначает значение типа *сообщение*; *почтовый_ящик* является именем буфера, используемого для связи двух процессов, а *p* — приоритет сообщения. Процесс может получить сообщение, обратившись к предопределенной процедуре

получить(*x, почтовый_ящик*)

Сам буфер организован в основном как очередь FIFO (первым пришел — первым ушел); но, с учетом приоритета, который специфицируется для каждого присылаемого в буфер сообщения, это упорядочение может переорганизовываться так, чтобы срочные сообщения проскакивали в начало очереди. Размер буфера выбирается с учетом необходимости удовлетворить различные требования. Если буфер заполнен, то обратившийся к процедуре *послать* процесс будет приостановлен до тех пор, пока некоторый другой процесс не удалит из буфера сообщение. Обычно размер буфера выбирается так, чтобы эта ситуация происходила как можно реже. Аналогично, если буфер пуст, то будет приостановлен процесс, обратившийся к процедуре *получить*; он будет ждать, пока в буфер не будет прислано сообщение. Задавая размер буфера равным 1, мы имеем возможность работать с примитивами передачи сообщений, обеспечивающими точно такой же эффект синхронизации, который предоставляют сигналы. Поэтому методом прохождения сообщений можно программировать наряду с взаимодействием и синхронизацию.

Это краткое описание приведено для того, чтобы показать главные идеи метода прохождения сообщений. В практических системах потребуются дополнительные возможности, такие, как возможность получения одного сообщения из множества сообщений с различных буферов (недетерминизм) и возможность ограничения доступа к буферу для специфицированных процессов. Впрочем, главное, что нужно не упустить из виду,— это то, что система прохождения сообщений является типичным примером практического решения вопроса взаимодействия процессов. Если взаимодействие процессов ограничивается передачей со-

общений, то все прекрасно. Впрочем, реализация системы прохождения сообщений является относительно дорогой с точки зрения работы готовой программы. Поэтому если механизм прохождения сообщений придется часто использовать для программирования других видов взаимодействия процессов, например для защиты общих ресурсов, то конечные программы будут трудны для понимания и мало эффективны.

6.7. ТРЕБОВАНИЯ К ЯЗЫКАМ РЕАЛЬНОГО ВРЕМЕНИ

В предыдущих разделах этой главы были описаны общие механизмы спецификации мультипроцессных систем. Языки реального времени требуют кроме этих механизмов и некоторые другие, специальные механизмы.

Одним из главных требований к большинству систем реального времени является способность обработки ситуаций, содержащих ошибки. Везде, где это только возможно, система должна уметь восстанавливаться после ошибок; подробно вопросы восстановления обсуждаются в гл. 8. Впрочем, в безвыходных ситуациях может оказаться необходимым прибегнуть к прекращению процесса. На уровне проекта языка такая возможность обеспечивается достаточно просто введением оператора прекращения. Задание оператора *прекратить*(*P*) внутри некоторого процесса Q ведет к прекращению процесса *P*. К сожалению, за видимой простотой оператора прекращения таится ряд проблем, которые нужно решать и программисту, вынужденному разбираться во всех последствиях применения этого оператора, и разработчику языка, который должен определять его семантику, и составителю компилятора, реализующему результат прекращения процессов.

Во-первых, следует заметить, что должны выполняться общие правила нормального завершения процессов, т. е. из факта прекращения одного процесса должно следовать, что прекращаются и все его потомки, созданные процессом прямо или косвенно. Во-вторых, для каждого прекращенного процесса, принимающего участие в межпроцессном взаимодействии, должны выполняться следующие действия. Если процесс находится в состоянии ожидания в очереди (например, в мониторе или ожидая рандеву), то его нужно удалить из очереди. Если процесс был фактически занят в операции общения, то должен быть послан сигнал об ошибке каждому процессу, на который может повлиять его внезапное завершение. Легко понять, что таким образом дерево прекращения процессов может распространиться очень далеко. Основные проблемы, с которыми сталкивается программист, состоят в том, что ему необходимо предвидеть все возможные последствия прекращения любого процесса и

обеспечить соответствующие механизмы восстановления. Это легко может привести к сверхусложненным проектам программ, которые трудно понимать, обосновывать и поддерживать. Реализация механизма прекращения также трудна, особенно если базовое машинное оборудование в действительности ориентировано на параллельность. Отсюда, следовательно, нужно сделать вывод, что, хотя оператор прекращения может оказаться и необходим, использовать его нужно только в самых крайних случаях.

Другие нужные для систем реального времени возможности связаны с синхронизацией во времени. Во-первых, должно иметься некоторое средство доступа к часам реального времени как для целей приостановки процесса на заданный период, так и для ожидания наступления конкретного момента времени. Этот вид возможностей можно обеспечить либо непосредственно в форме системных процедур, либо косвенно, предоставляя средства построения процессов-таймеров, которые предлагают такие услуги процессам пользователей.

Во-вторых, должно иметься некоторое средство для того, чтобы программировать условия завершения по времени межпроцессного общения. Чтобы проиллюстрировать один возможный способ решения проблемы завершения по времени, рассмотрим систему с моделью рандеву. В этой модели некоторый процесс может ждать обращения из некоторого множества обращений, включив операторы приема в оператор выбора. Простое обобщение механизма выбора позволяет программировать завершения по времени простым и естественным образом. Вводится оператор истечения[7], который может появиться в операторе выбора в качестве альтернативы. Оператор истечения имеет вид

> **after** *n*;
> **…**
> **end**

где *n* специфицирует время в терминах некоторой стандартной единицы (скажем, миллисекунды). Оператор истечения содержит тело, которое выполняется, если в течение *n* единиц времени не произойдет обращение к одной из остальных альтернатив приема в операторе выбора. Использование оператора истечения лучше всего показать на примере.

Предположим, что в системе при архитектуре с распределенными процессорами требуется процесс, который передавал бы сообщения от процессов в своей собственной вершине к процессам в некоторой другой удаленной вершине. Факт передачи каждого сообщения не гарантирует приема сообщения; после

---

[7] Со служебным словом **after** (после). — *Прим. ред.*

каждого переданного сообщения передающий процесс ждет получения подтверждения. Если он не получает подтверждения в течение одной секунды, он передает сообщение еще раз; прежде чем заявить об ошибке, он пытается повторно передавать сообщения не больше 5 раз. Ниже приводится эскиз такого процесса в командах. Предполагается, что процедуры *передать* и *подтвердить* обеспечиваются соответствующим оборудованием интерфейса.

```
process переправить;
  var m: сообщение;
     отправлено : логический;
    счетчик : целый range 0..5;
begin
   do
      accept послать(in этот : сообщение);
         m := этот  end;
      счетчик := 0; отправлено := ложь;
      while (счетчик < 5) and not отправлено do
           передать(m);
           select
             accept подтвердить;
               отправлено := истина
            end
           or
             after 1000; {мс}
               счетчик := счетчик + 1  end
             end
         end select
     end do;
     if not отправлено then ошибка связи end if
   end do
end;
```

Сообщение передается процессу *переправить* для передачи с помощью обращения

переправить.послать(*m*)

Процесс *переправить* передает затем сообщение и входит в оператор выбора, который содержит две альтернативы. В альтернативе истечения специфицировано завершение по времени в 1000 мс. Если в течение этого периода происходит обращение к процедуре *подтвердить, то* выбирается оператор приема *подтвердить,* в противном случае, когда заканчивается период завершения по времени, выбирается оператор истечения. Обращение к процедуре *передать* и оператор выбора выполняются в

цикле, что позволяет заново передавать сообщение до четырех раз, прежде чем отказаться от попытки успешной передачи и инициировать «ошибку связи».

Рассматриваемый подход к программированию завершения по времени в системе, базирующейся на механизме рандеву, гибок и понятен. И на самом деле оператор выбора является, по-видимому, естественной конструкцией обработки завершений по времени, который не имеет соответствующего эквивалента в других определениях межпроцессного взаимодействия. Например, при использовании мониторов реализация завершения по времени требует введения отдельных «сторожевых» процессов-таймеров. Они, как правило, трудны для разработки и несколько эвристичны по своей природе.

Этим обсуждением конкретных особенностей механизмов реального времени мы заканчиваем общее введение в разработку конструкций параллельного программирования. В следующем разделе этой главы рассматриваются различные вопросы реализации, в последнем разделе этой главы дается общий обзор параллельного программирования в практических языках. Более подробное описание проектов практических параллельных языков можно найти в гл. 12 и 13.

6.8. АСПЕКТЫ РЕАЛИЗАЦИИ

Как было отмечено в начале этой главы, некоторые элементы параллельного программного обеспечения необходимы при решении многих задач в области систем реального времени. Однако степень параллельности (т. е. количество процессов), которая вводится в проект системы, может, по-видимому, потребовать некоего компромисса между легкостью и ясностью программирования и эффективностью конечной реализации. Независимо от того, какую архитектуру имеет базовое машинное оборудование, однопроцессорную или мультипроцессорную, с большой уверенностью можно утверждать, что число процессов в машине превысит число имеющихся в наличии физических процессоров. Следовательно, обязательно потребуется организовать аппарат переключения процессоров на процессы. Такая мультиплексная организация требует, чтобы это зависящее от контекста переключение происходило каждый раз, когда выполняемый текущий процесс заменяется на другой процесс; ясно, что скорость выполнения рабочей программы уменьшится. Кроме того, разумеется, будут издержки, связанные с реализацией механизмов взаимодействия и синхронизации процессов.

В этом разделе мы не собираемся подробно описывать стратегии реализации; будет сделана попытка выделить те области, в которых возможна оптимизация, существенно уменьшающая

перечисленные издержки. В частности, будет показано, что, хотя модель рандеву с первого взгляда требует значительно больше процессов для реализации заданной системы, чем модель с монитором, на практике эти требования могут быть значительно снижены. Другими словами, язык программирования, который предоставляет для мультипрограммирования гибкие механизмы высокого уровня, не должен исключаться из рассмотрения только потому, что при поверхностном рассмотрении кажется, что он ведет к менее эффективным проектам. Наоборот, важнейшим требованием является как раз то, чтобы язык позволял выражать параллельные вычисления способом, непосредственно моделирующем требования задачи, и не заставлял бы программиста прибегать к неясным программным структурам из-за отсутствия в языке нужных для задачи конструкций. Только при наличии такого языка может быть написано легко понимаемое, легко поддерживаемое и надежное программное обеспечение.

Традиционным подходом к распределению ресурсов физического процессора среди некоторого числа процессов является разделение времени. Процессор соединяется с часами некоторым каналом прерывания. При каждом прерывании процессор временно прекращает выполнять текущий процесс и выбирает для выполнения некоторый другой[8] процесс. Кроме этого нормального контекстного переключения происходит также контекстное переключение каждый раз при приостанавливании текущего процесса (например, при обращении к мониторной процедуре). В системе реального времени взаимодействие процессов таково, что эта вторая причина контекстного переключения достаточна сама по себе для моделирования параллельного поведения. Следовательно, накладные расходы, связанные с разделением времени, могут быть существенно сокращены.

Непосредственным следствием этого вида оптимизации является то, что присущий уровню описания программы детерминизм существенно уменьшается на базовом уровне реализации. В результате при условии, что генерируемые машинным оборудованием контекстные переключения, определяемые операциями ввода-вывода, обрабатываются должным образом, становится также возможно существенно оптимизировать механизм взаимного исключения, такой, как монитор. Хорошим примером этого является стратегия реализации, принятая в языке Модула (см. гл. 12). В Модуле различаются процессы, которые взаимодействуют с машинным оборудованием (процессы устройств), и процессы, которые с ним не взаимодействуют (орди-

---

[8] А возможно, и тот же самый. — *Прим. ред.*

нарные процессы). Для защиты доступа к общим ресурсам со стороны ординарных процессов на уровне языка введены мониторы; однако в базовой реализации нет явного механизма взаимного исключения. Но его и не нужно просто потому, что стратегия формирования очереди обеспечивает невозможность для любых двух процессов одновременного доступа к мониторной процедуре (Вирт, 1977с).

Согласимся, что реализация Модулы является экстремальным примером такой оптимизации, которая все-таки ведет к некоторым ограничениям использования языка. Тем не менее она показывает, что тщательно разработанная система прохождения программы может существенно уменьшить кажущиеся неизбежными накладные расходы, связанные с мультипрограммированием.

Системы, разработанные с использованием модели рандеву, должны, по-видимому, характеризоваться дополнительным потреблением времени при прохождении программы в связи с необходимостью часто обращаться к помощи «третьих лиц» — процессов, обслуживающих асинхронные процессы. Ярким примером этого является процесс буферизации, описанный в разд. 6.5. Впрочем, в практической реализации компилятор вполне может удалить такие процессы, давая возможность вызывающим их процессам выполнять реализующие их команды линейным образом. Чтобы проиллюстрировать это, покажем, как можно обойтись без процесса буферизации. Обращение к

буферизация.передать(*x*)

заменяется на

```
L1: оградить(замок);  if not полон then
        занести(x); {тело оператора приема}
        освободитъ(замок);
        послать (завершен);
        goto L2
    end if;
    освободить(замок);
    ждать(завершен);
    goto LI
L2:
```

а обращение к

буферизация.получить(*x*)

заменяется на

```
L1: оградить (замок);
     if not пуст then
      выбрать(x); {тело оператора приема}
        освободить(замок);
        послать(завершен);
       goto L2
     end if;
     освободить(замок);
     ждать(завершен);
     goto LI
L2:
```

Здесь *замок* — защищающий буфер семафор, а *завершен* — сигнал, указывающий на завершение оператора выбора.

Этот вид оптимизации имеет смысл применять тогда, когда предохранители оператора приема в большинстве случаев открыты, а если они закрыты, то обращение к следующему оператору приема их откроет. Это, безусловно, верно для процесса буферизации и на самом деле верно для большинства процессов типа «третьих лиц».

Все рассмотренные оптимизации построены так, что обладают достаточно общей структурой и могут быть применены к любому промежуточному процессу. В частности, если попытка передать сообщение окончилась неудачно, то при повторной попытке предохранитель перевычисляется. Ясно, что в нашем случае это необязательно, так как процесс буферизации обладает тем свойством, что выполнение одного оператора приема всегда открывает предохранители другого. Следовательно, возможна более тонкая оптимизация; ее код был бы очень похож на код, использованный в решении 2 задачи поставщик/потребитель (разд. 6.2). Впрочем, такую оптимизацию может оказаться трудно получить автоматически в компиляторе.

Из приведенного выше обсуждения следует, что в мультипроцессных программах имеется несколько потенциальных областей для оптимизации. Практическая возможность оптимизации для монитора была продемонстрирована в реализации Модулы; но оптимизация рандеву более проблематична, так как пока что не существует реализующего ее практического компилятора. Впрочем, в теоретических работах возможность ее реализации была показана (Хаберман и Насси, 1980).

6.9. ПАРАЛЛЕЛЬНОСТЬ В ПРАКТИЧЕСКИХ ЯЗЫКАХ

Ранее в языках возможности параллельного программирования обычно обеспечивались многозадачной операционной системой (МЗОС). Интерфейс в МЗОС мог быть либо оставлен

неопределенным, либо стандартно определенным, но внешним по отношению к языку, либо определенным в самом языке. Первые версии языка CORAL 66 принадлежали первой категории (Вудвард и др., 1970), но в результате возникших в связи с отсутствием стандартизации проблем теперь в язык введено программное обеспечение параллельного программирования MASCOT (Симпсон и Джексон, 1979). Язык RTL/2 (Барнес, 1976 и гл. И) является примером языка второй категории. В нем нет явных языковых конструкций для мультипрограммирования, но для языка был разработан ряд стандартных операционных систем (ICI, 1973). Язык PEARL (Брандес и др., 1970) является примером языка третьей категории. В нем имеются языковые конструкции для запуска и остановки процессов регулярно через фиксированные интервалы времени и для программирования синхронизации и взаимного исключения с использованием сигналов и семафоров. На самом деле, язык PEARL воплощает всю операционную систему PEARL, в функции которой входит управление процессами, составление смеси задач, обработка прерываний, ввод-вывод, управление файлами и памятью. Одной из проблем, возникающих при реализованном в языке PEARL подходе, является некоторая необозримость языка, что привело к преимущественному использованию его подмножеств.

Первым практическим языком программирования, в котором появились мониторы, был параллельный Паскаль (Бринч Хансен, 1975а). С тех пор было создано еще несколько похожих языков; среди них нужно отметить языки Модула (Вирт, 1977а и гл. 12) и Паскаль Плюс (Уэлш и Бастард, 1979). Из проектов «особо надежных» языков мониторы включают как Red (Нестор, 1978), так и Yellow (SRI, 1978), хотя, как это ни странно, язык Blue (Softech, 1978) предоставляет только семафоры и сигналы.

Пионерская работа Хоара, заложившая основы модели рандеву для взаимодействия процессов (Хоар, 1978), привела к созданию языка CSP (Взаимодействующие Последовательные Процессы), однако до настоящего времени имелись только экспериментальные реализации этого языка. Проект языка Green (Ихбиа, 1978) был тесно связан с языком CSB, но в последующем проекте языка Ада (см. гл. 13) была произведена некоторая ревизия конструкций CSP, выразившаяся в том, что явные операции ввода-вывода языка CSP были заменены на операторы приема процедурного типа, аналогичные операторам приема, с которыми мы познакомились в этой главе.

И в заключение следует заметить, что здесь мы обсудили не все возможные подходы к построению параллельных языков. В частности, следующие два представляются достаточно важны-

ми и заслуживающими упоминания. Во-первых, выражения для путей доступа (Кэмпбелл и Хаберман, 1974) являются механизмом, с помощью которого доступ к структурам мониторного типа управляется регулярными (путь доступа) выражениями, которые специфицируют порядок возможных доступов. Во-вторых, в подходе управление/данные обусловленное данными взаимодействие процессов заменяется на явное управление процессами. Система специфицируется как набор процессов, для которого указаны пути прохождения данных (граф данных); имеется также управляющая структура (граф управления), которая точно указывает, когда должен выполняться каждый процесс (см., например, Даглес и Даусинг, 1979).

Глава 7.

ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА НИЗКОГО УРОВНЯ

7.1. ВВОД-ВЫВОД В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

Из большого количества специфических особенностей, которые отличают систему реального времени от других базирующихся на вычислительных машинах систем, важное значение имеет то уникальное взаимоотношение, которое имеется между вычислительной машиной и внешней средой. В самом деле, для вычислительной машины, работающей в реальном времени, синхронизация ее деятельности с внешними стимулами и общение с большим разнообразием «странных» устройств ввода-вывода являются необходимостью, которая ставит эту машину в особое положение. Из этого вытекает, что в языке реального времени должны иметься возможности для непосредственного программирования устройства машинного оборудования низкого уровня. Прежде чем рассмотреть формы, в которых должны выражаться эти возможности, будет полезно сделать общий обзор методов, которые обычно применяются в существующих языках и системах при обработке ввода-вывода.

В вычислительной системе общего назначения пользователь всегда вынужден выполнять операции ввода-вывода, пользуясь только стандартными периферийными устройствами (такими, как печатающие устройства, терминалы, диски и т. д.). Вследствие этого разработчики языков общего назначения имеют возможность предоставить пользователю стандартный интерфейс ввода-вывода высокого уровня, часто в форме предопределенных библиотечных процедур. Общераспространенным методом

является трактовка ввода-вывода как потока символов, который может быть направлен в конкретное устройство управляющей программой. Затем предоставляются процедуры для преобразования из внутреннего представления данных во внешнее представление в символьном формате и обратно, а также для передачи одиночных символов и блоков символов между определенными устройствами. Внутренняя структура этих операций ввода-вывода, однако, совершенно скрыта от пользователя. Главные компоненты подсистемы ввода-вывода являются обычно частью операционной системы, и процедуры ввода-вывода языка программирования играют роль простого интерфейса. Таким образом, у пользователя нет возможности перепрограммировать операции ввода-вывода так, чтобы они лучше соответствовали его нуждам. Когда возникает необходимость внести изменения в систему ввода-вывода (например, при установке нового устройства), обращаются к специальному системному программисту, чтобы он внес изменения в операционную систему. В случае нового устройства ввода-вывода это потребует включения нового драйвера, код которого большей частью будет написан на языке ассемблера.

Итак, вычислительные системы общего назначения характеризуются относительно статическими конфигурациями машинного оборудования. Пользователи, как правило, не собираются программировать свои собственные устройства ввода-вывода специального назначения, и поэтому языки программирования общего назначения не предоставляют для этого средств. Относительно высокая стоимость разработки первоначального комплекса программного обеспечения и его дальнейшего поддержания часто может быть распределена на достаточно большое число установок; каждая новая установка требует лишь второстепенных изменений в программном обеспечении для удовлетворения нужд конкретной конфигурации машинного оборудования.

В отличие от этого свойства системы общего назначения системы реального времени обладают совершенно иным множеством характеристик. Типы устройств ввода-вывода, требующих связи с работающей в реальном времени вычислительной машиной, разнообразны и многочисленны. Наряду со стандартными периферийными устройствами может потребоваться связь с цифровыми устройствами ввода-вывода, с аналого-цифровыми преобразователями, с устройствами графического вывода, с приборным интерфейсом и т. д. Времена ответа вычислительной машины на внешние события (например, прерывание по времени, сигнализирующее о необходимости просмотреть вводные данные от аналого-цифровых преобразователей) гарантированно должны укладываться в заданные пределы, характерные только для

данной конкретной системы. Из этого следует, что для системы реального времени нет какой-то одной фиксированной схемы и очень часто приходится разрабатывать программное обеспечение для конкретной уникальной конфигурации, не надеясь распределить стоимость его создания между несколькими установками. Кроме того, следует заметить, что особого внимания к себе операции ввода-вывода требуют еще и потому, что часто стирается разница между прикладной программой и операционной системой. Многие компоненты традиционных операционных систем, такие, как подпрограммы управления устройствами ввода-вывода, становятся частью прикладной программы. В действительности очень часто оказывается более точным рассматривать систему реального времени как набор прикладных программ, которые работают непосредственно на «голом» машинном оборудовании (или по крайней мере на очень небольшом ядре операционной системы), а не как традиционную структуру из прикладных программ большой операционной системы общего назначения и машинного оборудования (см. рис. 7.1).

Из этих характеристик систем реального времени следует, что в отличие от языков общего назначения язык реального времени должен предоставлять средства для непосредственного программирования устройств ввода-вывода самых необычных видов, удовлетворяя индивидуальные требования системы.

Принимая во внимание очевидную необходимость иметь средства программирования устройств, интересно отметить, что большинство существующих языков реального времени предоставляют для этого ничтожно мало возможностей. Очень часто главным средством является возможность использовать наряду с кодом высокого уровня и код на языке ассемблера. Кроме того, операционные системы реального времени разрабатываются так, чтобы к ним можно было «легко» добавлять новые подпрограммы управления устройствами. Самое лучшее, что можно сказать про такие возможности, это то, что они неадекватны.

В данной главе описаны возможности, которые позволяют выполнять программирование устройств низкого уровня непосредственно на языке высокого уровня. В этой области, как и в остальных, мы хотим получить возможность писать ясные, надежные и легко поддерживаемые программы, затрачивая на это минимальные усилия. Изложение состоит из трех частей. В следующем разделе рассматривается доступ к машинному оборудованию, затем в разд. 7.3 обсуждается подходящая модель для программирования устройств ввода-вывода. Введя основные языковые возможности, в разд. 7.4 мы иллюстрируем их использование на примерах программирования типичных операций ввода-вывода.

В отличие от большинства других областей разработки языка программирования здесь невозможно стремиться к достижению полной машинной независимости. Вместо этого делается попытка ввести самые общие понятия, которые легко можно

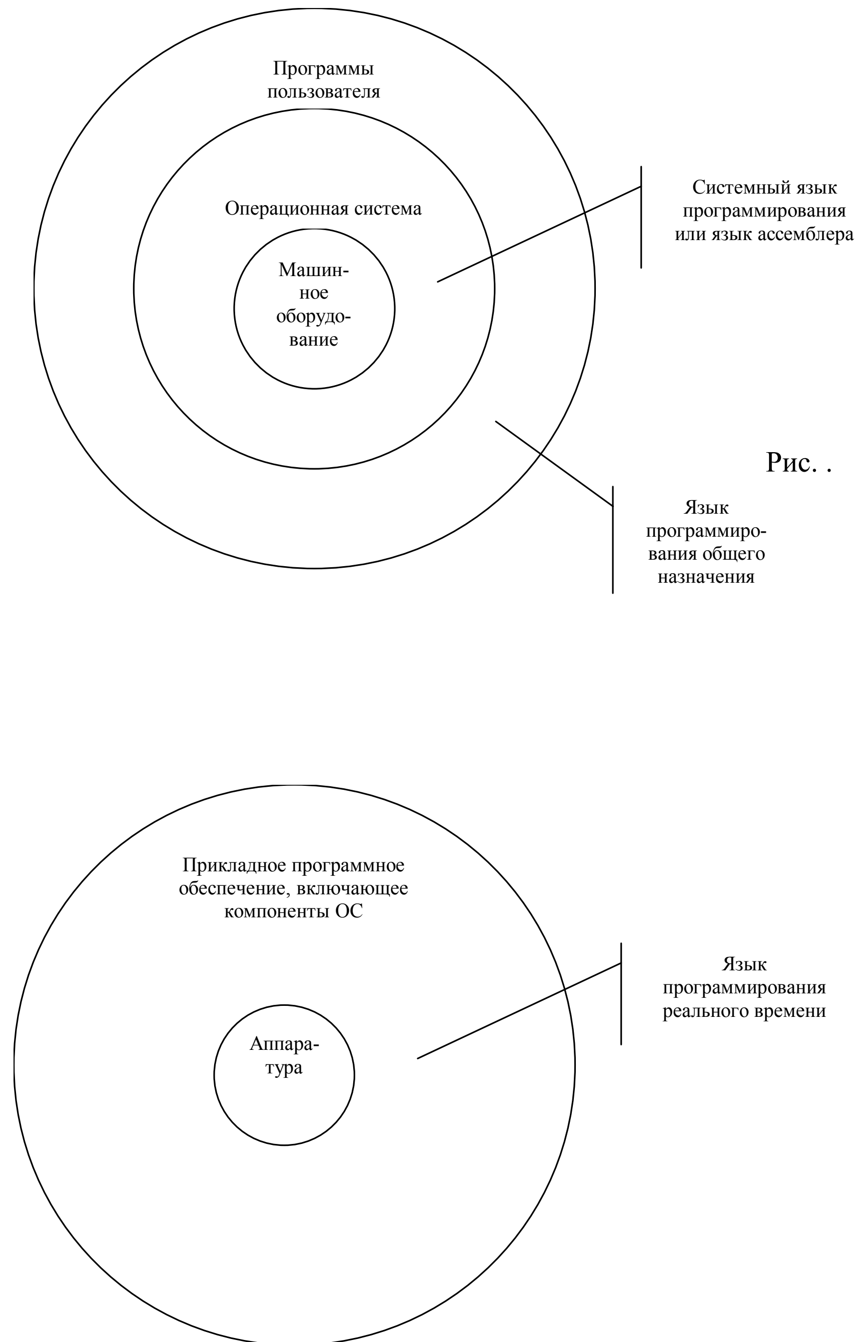


Рис 7.1. Структура вычислительных систем (a) общего назначения и (b) реального времени.

было бы приспособить к широкому спектру конфигураций машинного оборудования. Конкретные примеры мы будем рассматривать на двух машинных архитектурах: на PDP 11 фирмы Digital Equipment Corporation (DEC, 1972) и на вычислительной машине Intel 8020 (Intel, 1977). Они были выбраны главным образом потому, что широко известны. Впрочем, следует признать, что архитектура PDP 11 особенно удобна для демон-

страции того общего подхода к программированию ввода-вывода, который мы собираемся здесь защищать (подробное обсуждение этой темы см. в работе (Хант, 1980)).

7.2. ДОСТУП К МАШИННОМУ ОБОРУДОВАНИЮ

Доступ к машинному оборудованию из языка высокого уровня зависит от наличия у программиста механизма явной спецификации отображения абстрактных типов данных языка на физическую память и регистры ввода-вывода базовой машины. В слабо типизируемом языке это не представляет особенной проблемы. Как было сказано в гл. 2, простое целое можно рассматривать как битовую строку фиксированной длины. При задании множества операций «битового маневрирования», таких, как логические команды, команды сдвига и циклического сдвига, появляется возможность работать с индивидуальными битами и битовыми полями управляющих регистров и регистров статуса устройств ввода-вывода. Главный недостаток этого подхода состоит в том, что во многих отношениях он немногим лучше программирования на языке ассемблера, так как получающиеся в результате программы трудны для понимания, верификации и сопровождения. Более того, следуя этому подходу, мы теряем присущую сильной типизации надежность. В данном разделе будет показано, как можно использовать описанные в гл. 2 и 3 типы данных в программировании низкого уровня на устройства ввода-вывода, чтобы обеспечить надежный доступ к периферийным регистрам, не обращаясь к слишком детальным операциям манипулирования с битами.

7.2.1. Адресация регистров ввода-вывода

Обычно при описании переменной, такой, как, например,

var *c*: *сим*;

компилятор автоматически выделяет для нее место в памяти и программисту не важно, где именно фактически расположена эта переменная. Но при программировании низкого уровня для ввода-вывода должна иметься возможность описать переменную с указанием конкретного адреса, так чтобы обеспечить доступ к отображенным на память регистрам ввода-вывода.

Это требование достаточно тривиально; можно удовлетвориться простым расширением синтаксиса описания переменной за счет части **at**. Например,

var *кбс* at #177562 : *сим*;

описывает переменную *кбс* типа *сим,* отображенную на ячейку памяти # 177562 (где # обозначает восьмеричное число), что на PDP 11 является панельным буферным регистром. Используя описание переменной *кбс,* можно выводить на панельный регистр символы с помощью простого присваивания. Например, *кбс :='9'* перешлет символ '*9'*. Заметим, что правила сильной типизации сохраняются несмотря на то, что *кбс* представляет регистр машинного оборудования; поэтому ошибки пробивки, такие, как отсутствие кавычек, будут обнаружены компилятором в обычном порядке.

Для работы на машинах, на которых должны специфицироваться составные адреса, включающие такие компоненты, как *страница* и *приписка,* достаточно разрешить употреблять за ключевым словом **at** значения любого типа. Например, при наличии

```
type адр = record
              страница : целый range 0..255;
              приписка : целый range 0..4095
           end;
```

любая переменная может быть отображена на конкретный адрес с помощью описания

```
var x at адр (0, 20): некоторый_тип;
```

Другими словами, структура адреса идентифицируется как тип данных, и на различные машинные регистры ссылаются затем по значениям этого типа. На практике такие типы будут обычно предопределяться при реализации языка на конкретную машину.

Для машин, на которых имеются как отдельно адресуемые части ввода-вывода, так и отображенные на память регистры ввода-вывода (например, Intel 8080), потребуется несколько расширить данный механизм. Переменная может быть отображена на элементы ввода-вывода при включении ключевого слова **port** в описание переменной. Например,

```
var упрр at port $DF : режим_таймера
```

описывает переменную *упрр,* соответствующую элементу вводавывода по адресу *$DF* ($ обозначает шестнадцатеричное число), являющемуся 8-битовым регистром управления режимом программируемого таймера 8253 на однопанельной вычислительной машине Intel 8020. В данном случае присутствие ключевого слова **port** означает, что вычислительная машина при доступе к переменной *упрр* должна использовать команды ввода-вывода, а не команды ссылки на память.

7.2.2. Отображение определенных пользователем типов на машинные регистры

Отображение простых типов, таких, как *сим,* на символьные буферные регистры, аналогичные регистрам из разобранного примера, осуществляется самым непосредственным образом, так как у рассмотренного типа нет никакой внутренней структуры. Очень часто в аппаратуре ввода-вывода регистр разделяется на поля. Например, 8-битовый регистр режима таймера

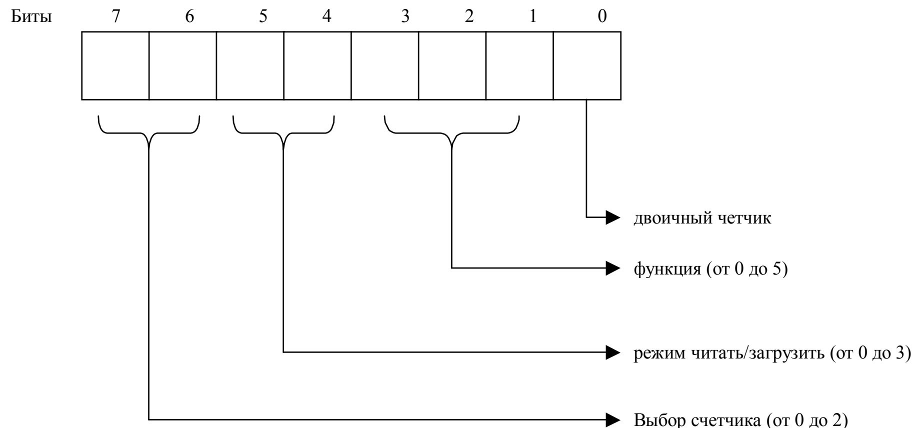


Рис. *7.2.* Управляющий регистр таймера 8253 системы Intel.

8253, который мы упоминали ранее, разделяется на четыре поля так, как показано на рис. 7.2. Регистр такого вида наиболее естественным образом представляется с помощью комбинированного типа данных, модифицированного так, чтобы разрешить явную спецификацию отображения каждой компоненты записи. И здесь для этого в описание типа записи может быть введена часть **at.** Например, тип *режим_таймера* может быть определен, следующим образом:

```
type режим_таймера = record
                        рсч at 0 : режим_счет;
                        фун at 1.. 3:функция_таймера;
                        рчз at 4..5 : чзрежим;
                        дсч at 6..7 ; целый range 0..2
                     end;
```

Из описания типа *режим_таймера* видно, что он содержит четыре компоненты; положение каждой компоненты специфицировано в записи указанием принадлежащих компоненте битов. Следовательно, например, компонента *рсч* представлена битом 0, *фун* представлена битами от 1 до 3. Предполагается, чтоотсчет битов ведется с 0, начиная с самого правого бита первого слова записи. Поэтому, если запись отображается на *m* n-битовых слов, то последний бит записи будет иметь номер
m X n – 1.

При программировании ввода-вывода часто бывает так, что в некотором конкретном поле регистра содержится значение статуса или значение режима. Например, полю режима читать/загрузить таймера 8253 должно быть присвоено значение из диапазона от 1 до 3 со следующим смыслом:

0 — операция запирания стетчика
1 — старший байт читать/загрузить
2 — младший байт читать/загрузить
3 — оба байта читать/загрузить

Наиболее очевидным образом описать тип *чзрежим* можно так:

type *чзрежи* : *целый* **range** 0..3;

но тогда такое присваивание, как *рт.рчз* : = 1, где *рт* - переменная типа *режим_таймера,* не будет содержать указания на то, для чего, собственно говоря, предназначался данный оператор присваивания. Ясно, что значения различных классов гораздо лучше были бы определены как перечисления, где было бы дополнительно указано базовое представление каждого перечисления, например:

type *чзрежим* = (*зап_дсч* => 0, *чз_стар* => 1,
чз_мл => 2, *чз_оба* => 3);

где => обозначает отображение абстрактного значения на базовое представление. Тогда приведенный выше оператор присваивания принимает вид:

рт.рчз := *чз_стар*

и теперь более понятно, что имеется в виду (см. пример в разд. 7.3.3, где даются определения других компонент типа *режим_таймера*) .

И наконец, программируя регистры устройств, которые содержат только индивидуальные биты статуса и управления, можно использовать множественный тип данных, который был описан в гл. 3. На базовой машине множества представляются битовыми строками. Поэтому все, что требуется, чтобы воспользоваться множественной нотацией при программировании регистров устройств этого типа, — это специфицировать, какой бит нужно взять для представления каждого элемента базового типа. Так, например, регистр статуса передачи последовательного линейного интерфейса PDP 11 содержит четыре бита статуса,

которые с помощью множественного типа можно представить следующим образом:

type *pen* = **set of** (*прервать* **at** 0, *продолжать* **at** 2, *невозможно* **at** 6, *готово* **at** 7);

здесь по-прежнему ключевое слово **at** используется для спецификации положения битов, на которые должны отображаться значения базового типа. Теперь для установки и проверки условий статуса можно использовать обычные применимые к множествам операции. Например, при задании описания

var *pcny* at #*1*77564 : *pcn*;

которое определяет регистр статуса печатающего устройства PDP 11, можно выполнять прерывания с помощью:

pcny := *pcn*(*невозможно*)

и очистку с помощью:

peny :=*pcn*( )

Проверять содержимое битов можно с помощью операции принадлежности множеству **in,** например,

if *готово* **in** *pcny* **then ...**

проверяет, установлен или нет бит *готово.*

Приведенное расширение типов данных, введенных в гл. 2 и 3, позволяет использовать их в области программирования устройств низкого уровня, включая дополнительную информацию для спецификации явного отображения на машинное оборудование. Во многих случаях текстуальное описание для переменных и типов устройств будет длинным, но это приемлемо и, вообще говоря, желательно, так как кроме того, что обеспечивается присущая правилам сильной типизации высокая надежность, использование идентификаторов в качестве имен компонент перечисления и записей вместо малоприятного битового кода вносит в ненадежную и трудную область ввода-вывода элементы удобочитаемости (и, следовательно, сопровождаемости).

7.2.3. Модули ввода-вывода

Программа, которая связана с машинным оборудованием непосредственно, используя описанные выше конструкции, безусловно должна содержать зависящие от оборудования и, следовательно, немобильные разделы. Поэтому следует обратить внимание на то, чтобы не нарушилась мобильность всей системы программного обеспечения из-за наличия в ней нескольких немобильных разделов. В гл. 5 была введена кон-

струкция модуля как средство объединения множеств соотносящихся объектов, для которой определялся интерфейс с внешней средой. Проблемы мобильности могут быть существенно упрощены, если требовать, чтобы все конструкции языка, ориентированные на ввод-вывод, могли употребляться только внутри модулей специального вида, называемых модулями ввода-вывода. В результате при переносе программного обеспечения (по крайней мере в принципе) переписывать нужно только код внутри этих модулей. Более того, так как эти модули написаны на языке высокого уровня, а не на языке ассемблера, эта перепись не потребует слишком много усилий. Примеры использования модулей ввода-вывода можно найти в следующих разделах.

7.3. МОДЕЛЬ ДЛЯ ПРОГРАММИРОВАНИЯ ВВОДА-ВЫВОДА НИЗКОГО УРОВНЯ

7.3.1. Механизмы машинного оборудования

На уровне машинного оборудования при осуществлении операции ввода-вывода выполняется ряд информационных обменов между двумя устройствами, работающими в существенно асинхронном режиме, т. е. между центральным процессором (ЦП) и устройством ввода-вывода. При обмене информации используется буфер данных, который очень часто является общим для всех устройств ввода-вывода. Поэтому первым важным требованием при программировании ввода-вывода является возможность адресации отдельных регистров конкретных устройств ввода-вывода; подробно это требование мы разобрали в разд. 7.2.1.

Чтобы модель, на которой мы собираемся отрабатывать абстрактные особенности языка программирования, удовлетворяла фактическим механизмам машинного оборудования, нужно прежде всего учесть тот важый факт, что устройства ввода-вывода могут работать параллельно с аппаратурой ЦП. Ранние машинные архитектуры не обладали этой возможностью; ЦП вынужден был выполнять цикл ожидания до тех пор, пока устройство ввода-вывода не закончит обработку данных. Статус этого устройства определялся ЦП, который постоянно обновлял признаки статуса в регистре статуса устройства. Поэтому этот вид операции ввода-вывода называется обновляемым режимом.

Недостаток обновляемого режима заключается, конечно, в том, что ЦП напрасно тратит время, пока выполняется каждая операция ввода-вывода. Чтобы ликвидировать эту причину неэффективности, был введен специальный механизм в форме механизма прерываний. Работая в режиме прерываний от вво-

да-вывода, ЦП инициирует операцию ввода-вывода и затем продолжает выполнять другие задачи. Устройство ввода-вывода работает параллельно с ЦП, а закончив выполнение своей операции, сигнализирует об этом ЦП посредством прерывания. Механизм прерывания выполняет две функции. Во-первых, он обеспечивает сохранность контекста прерванной программы. Во-вторых, он заставляет ЦП выполнить обслуживающую прерывание подпрограмму для всех операций завершения, которые требует устройство прерывания. Когда эта обслуживающая подпрограмма завершает свою работу, восстанавливается контекст прерванной программы.

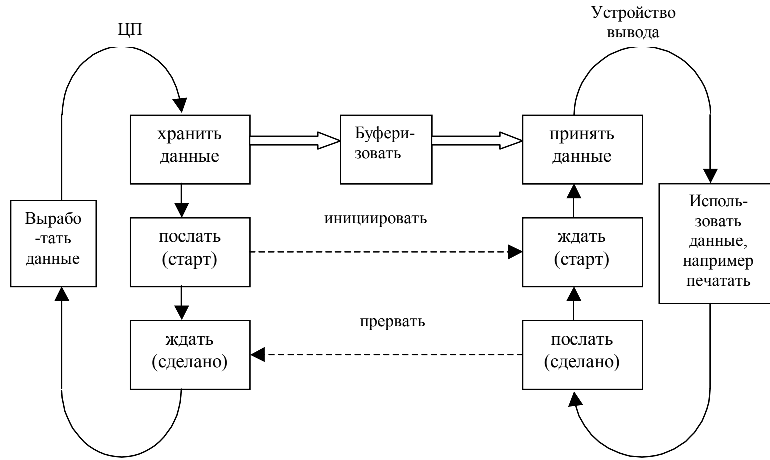
Механизм прерываний обеспечивает средство автоматического составления «протокола встречи», необходимого для взаимодействия асинхронных ЦП и устройства ввода-вывода. Фактическая передача данных должна, впрочем, по-прежнему выполняться под управлением ЦП. В тех случаях, когда нужно передавать очень большие объемы данных, может быть использован более тонкий аппаратный механизм, называемый механизмом прямого доступа к памяти (МПДП). При работе с этим механизмом ЦП сообщает устройству МПДП адрес памяти, куда (или откуда) нужно передать блок данных, и количество слов в блоке. Далее устройство МПДП выполняет передачу данных независимо от действий ЦП, «незаметно заимствуя» у ЦП циклы обращения к памяти (например, выполняя свой рабочий цикл каждый раз, когда ЦП выполняет внутреннюю операцию на сумматоре). Только после передачи всего блока данных устройство МПДП сигнализирует о завершении прерыванием ЦП.

Итак, механизмы машинного оборудования для передачи данных ввода-вывода по существу принципиально можно различать с помощью двух основных характеристик. Во-первых, требуемый протокол ввода-вывода может реализовываться или программным обеспечением с помощью метода обновления или машинным оборудованием посредством механизма прерывания. Во-вторых, передача данных может производиться последовательно по элементам данных под управлением ЦП или по блокам данных под управлением специального устройства МПДП. В этом отношении в системах реального времени наиболее часто встречающейся комбинацией являются, безусловно, передачи одиночных элементов данных под управлением с помощью прерываний; этот аппаратный механизм мы берем за основу абстрактной модели, которую будем строить в следующем разделе.

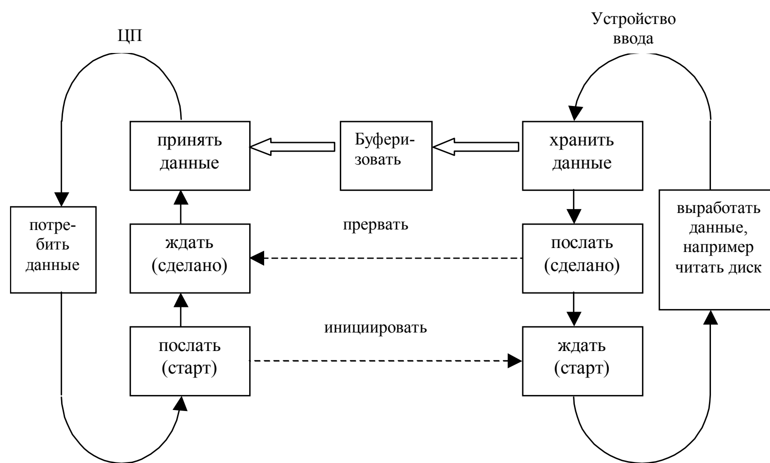
7.3.2. Абстрактная модель

Выполняемая устройством ввода-вывода деятельность может рассматриваться как деятельность процессора, выполняющего

некоторую фиксированную задачу. Поэтому модель для описания программирования устройства ввода-вывода должна обеспечивать абстрактное представление одного или более периферийных процессоров, работающих параллельно с центральным процессором. Ясно, что на уровне языка понятие процесса, ко-



a



b

Рис 7 3 Операции ввода-вывода при управлении с помощью прерывания (a) Вывод (b) Ввод.

торое мы обсудили в предыдущей главе, является абстракцией, удобной для представления системы этого вида.

Если считать, что деятельность ЦП представлена процессом и деятельность устройства ввода-вывода также представлена процессом, то операция чтения или записи с регистра или на регистр устройства соответствует межпроцессной связи низкого уровня. Так как синхронизация процессов реализуется с помощью прерываний, то прерывание соответствует сигналу машинного оборудования. Стоит заметить, что название прерывание пред-

полагает, что оно играет несколько иную роль, чем простой сигнал. На самом деле это название предполагает наличие модели, не обладающей никакой формой мультипроцессной структуры, поэтому прерывание следует рассматривать как механизм моделирования асинхронных деятельностей. То, что в результате такая модель оказывается сложной, объясняется в большей степени отсутствием средств программирования устройств низкого уровня в традиционных последовательных языках реального времени. По-видимому, только язык ассемблера обладает достаточной гибкостью для поддержания такой модели.

В рамках мультипроцессной системы результирующая модель с управляемыми прерыванием операциями ввода-вывода может быть изображена с помощью блок-схемы так, как показано на рис. 7.3.

ЦП и периферийное устройство работают параллельно, используя сигналы для синхронизации своей деятельности. Сигнал старт представляет действия ЦП по инициации устройства ввода-вывода, а сигнал сделано представляет генерацию устройством ввода-вывода прерывания для указания завершения операции. На практике, ожидая сигнала сделано, ЦП не бездействует, а выполняет некоторый другой процесс.

7.3.3. Модель языка

В предыдущей главе были представлены мультипроцессные языковые конструкции, причем особое внимание уделялось надежным механизмам межпроцессной связи. Два таких механизма описаны подробно: монитор и рандеву. При оценке с самых общих позиций второй был признан значительно более удобным. Одна из причин этого состоит в том, что, хотя монитор и является механизмом высокого уровня для доступа к общим ресурсам с взаимным исключением, он все же использует для синхронизации сигналы, т. е. элементы низкого уровня. Как это ни парадоксально, этот очевидный недостаток монитора превращается в его достоинство в области программирования низкого уровня, где прямое соответствие между прерываниями и сигналами позволяет построить более естественную модель и более эффективную реализацию. Поэтому здесь для иллюстрации того, как можно выполнять программирование низкого уровня на языке высокого уровня, мы будем пользоваться мониторной моделью. Однако из этого не нужно делать вывод, что понятие рандеву в этой области не может быть использовано. Его можно применить, если рассматривать прерывания как обращения машинного оборудования, к операторам приема, но такой способ кажется менее естественным и здесь рассматриваться не будет. Как было сказано ранее, зависящие от устройства языковые конструкции должны быть объединены в модуле ввода-

вывода. Форма модуля ввода-вывода очень похожа на форму обычного модуля, который был описан в гл. 5, и имеет вид:

device module *имя_мвв*[*p*];
  define ...
  use ...
private
  {реализационная часть}
end module *имя_мвв*;

причем единственным существенным отличием в синтаксисе является присутствие ключевого слова **device** (устройство) и заключенного в квадратные скобки целого значения *p*. Специфицируемое как *p* значение обозначает приоритет по прерыванию всего заключенного в модуле кода. Из этого следует, что только те прерывания, приоритет которых выше данного уровня, могут прервать выполнение кода модуля. У модуля ввода-вывода имеются еще два специальных свойства. Во-первых, он обладает тем свойством монитора, что если он экспортирует процедуры, то они могут быть выполнены другим процессом только при взаимном исключении. Во-вторых, как и монитор, модуль ввода-вывода может обеспечить интерфейс между взаимодействующими процессами. Впрочем, в этом случае один из взаимодействующих процессов будет всегда иметь специальный статус принадлежности к процессам устройств. Поэтому имеет смысл потребовать, чтобы такие процессы всегда описывались в модуле ввода-вывода для подчеркивания их специального статуса. В примерах из этого и последующих разделов данные особенности будут освещены более четко.

В качестве примера модуля ввода-вывода и изолированного процесса устройства рассмотрим задачу об использовании таймера 8253 для реализации часов реального времени. Интерфейс для этого модуля осуществляется с помощью сигнала программного обеспечения с именем *такт,* который посылается через каждую секунду ко всем ждущим его процессам. Сигнал *такт* генерируется процессом устройства *таймер,* который сам синхронизуется с реальным временем с помощью сигнала машинного оборудования (т. е. прерывания), генерируемого устройством 8253. Этот сигнал описывается следующим образом:

var *таймер_прер* **at** $3*FE*1 : *сигнал*;

где в части **at** специфицируется адрес вектора прерываний, соотносящийся с базовым прерыванием. Процесс *таймер* выполняет следующий бесконечный цикл:

```
do
   установить режим таймера;
   загрузить счетчик;
   ждать прерывания при исчерпании счетчика;
   послать такт ко всем ждущим процессам
end do
```

Теперь можно привести полный текст модуля ввода-вывода *часы*:

```
device module часы[6];
define такт : сигнал;
private
   const счетчик = 1000;
   type режим_счетчика = (двоичный => 0, dec => 1);
        функция _таймера = (счетчик =>0, одиночный => 1 ,
         скорость_ген => 2, прямоуг => 3,
         программный_строб => 4, машинный_строб => 5);
       чзрежим = (замок_дсч => 0, чз_стар => 1,
                 чз_мл => 2, чз_оба => 3);
        режим_таймера = record
                              рсч at 0 : режим_счет;
                              фун at 1..3 :
                              функция_таймера; рчз at 4..
                              5 : чзрежим;
                              дсч at 6. .7 : целый range 0..2
                           end;
  var такт : сигнал;
  process таймер;
  var таймер _прер at $3FE1 : сигнал;
      всчр at port $DF : режим_таймера;
      свеч at port $DD> : целый range 0..255;
  begin
   do
     всчр := режим_таймера(двоичный, счетчик, чз_оба, 1);
     свеч := счетчик mod 256; свсч := счетчик div 256;
     ждать (внутренний_таймер);
     while ожидаемый(такт) do
           послать(такт)
     end do
   end do
  end;
begin init таймер
end module часы;
```

На первый взгляд спецификация данного модуля *часы* кажется чрезвычайно многословной. Однако нужно осознать, что описания типа являются в конечном счете достаточно полным описанием интерфейса программного обеспечения для таймера 8253. Поэтому, когда программисту понадобится модифицировать модуль часов, ему не придется справляться о руководством,

чтобы узнать, что означают различные биты регистра управления; все это для него заготовлено в спецификациях модуля. Чтобы подчеркнуть это, рассмотрим альтернативное описание управляющего регистра режима таймера:

type *режим_таймера* = *целый* **range** 0..255; и соответствующую

операцию установки

всчр := $0*D*

которая непонятна, если не проконсультироваться с руководством по программированию устройства 8253.

Процесс устройства *таймер* по своей функции практически эквивалентен подпрограмме обслуживания прерываний на языке ассемблера. Хотя концептуально его можно представить работающим параллельно с другими процессами в системе, на практике он выполняет каждый раз при поступлении прерывания таймера только один цикл. Это предоставляет компилятору большую свободу для оптимизации при составлении очереди процессов устройств, что позволяет достичь производительности программы, сравнимой с производительностью при эквивалентной обработке прерываний, закодированной на языке ассемблера.

7.4. ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА

В данном разделе будет приведен ряд примеров, иллюстрирующих использование описанных модели и языковых конструкций в типичных ситуациях программирования ввода-вывода. Все примеры для конфигурации машины PDP 11, но, как правило, применимы и для большинства других используемых в настоящее время архитектур небольших компьютеров и микрокомпьютеров.

7.4.1. Передача одного символа данных

Простейший способ передать один элемент данных — это написать модуль ввода-вывода с одной процедурой интерфейса, которая непосредственно обращается к устройству машинного оборудования так, как показано на рис. 7.4. Конкретный пример такого обращения имеет вид:

```
device module вывод_один_сим[4];
   define procedure вывсим(in c : сим);
private
   type срс = set of (невозможно at 6, готово at 7);
   procedure вывсим(in c : сим);
   var прер at #64 : сигнал;
     статус at #77564 : срс;
     буфрег at #177566 : сим;
begin
   буфрег := с; {поместить символ}
   статус := срс(невозможно);{разрешить
                                        прерывание}
   ждать(прер); {ждать прерывание}
    статус := срс( ) {снять прерывание}
end;
begin
end module вывод_один_сим;
```

Модуль ввода-вывода *вывод_один_сим* содержит процедуру *вывсим,* к которой может обратиться любой процесс, чтобы передать одиночный символ на печатающее устройство PDP 11.

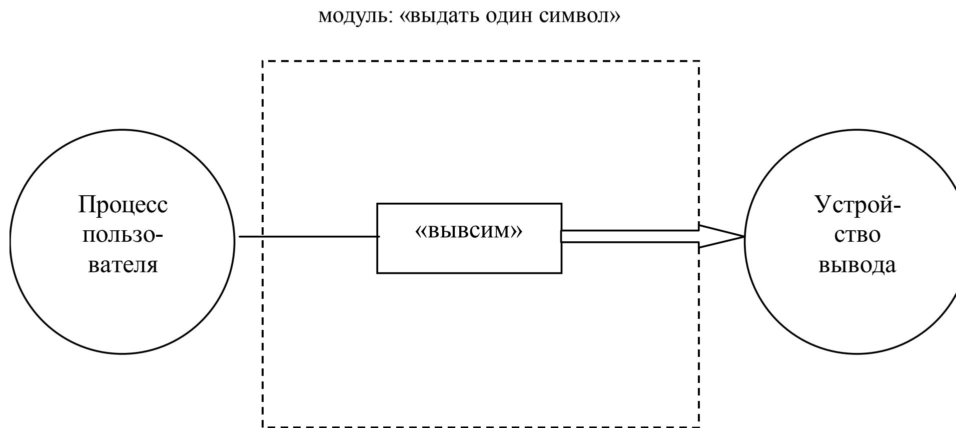


Рис. 7.4. Простая выдача одного символа.

Работа этой процедуры очень проста. Символ, который нужно передать, помещается в буферный регистр печатающего устройства, разрешается прерывание и затем вызывающий процесс приостанавливается с помощью команды ожидания до тех пор, пока не завершится передача. Это в точности соответствует модели, изображенной на рис. 7.3а, где сигнал старт задается неявно операцией записи в буферный регистр печатающего устройства.

Такая же схема может быть использована для обновляемого режима ввода-вывода, процедура *вывсим* в этом случае принимает следующий вид:

```
procedure вывсим(in c : сим);
   var статус at #177564 : срс;
   буфрег at #177566 : сим;
 begin
   буфрег := c;
   while not(готово in статус) do
     {безразличное действие}
    end do
 end;
```

но теперь вызывающий процесс не приостанавливается, ожидая сигнала, а выполняет некоторый цикл.

Недостатком данного подхода является то, что вызывающая процедура приостанавливается на время выполнения передачи

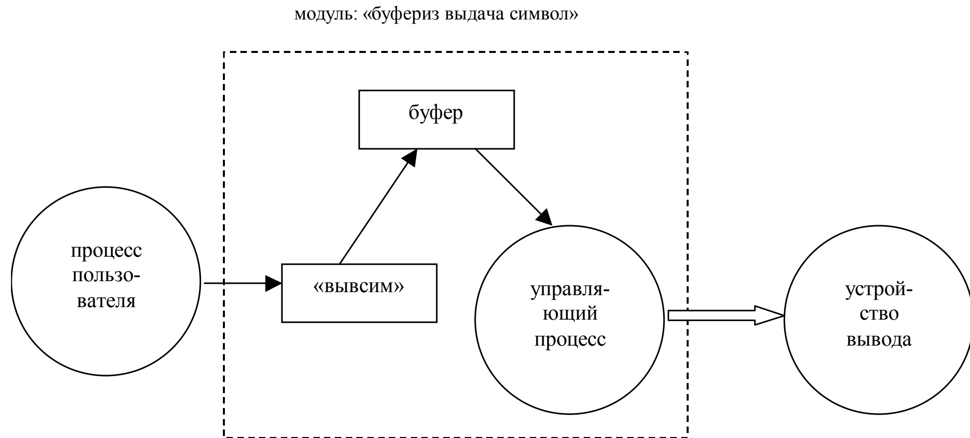


Рис. 7.5. Буферизованный вывод одного символа.

ввода-вывода. Этого можно избежать, если отделить вызывающий процесс от аппаратуры ввода-вывода введением промежуточного процесса устройства, связанного с помощью буфера, как показано на рис. 7.5. В результате мы получаем модуль ввода-вывода примерно такого вида:

```
device module буфериз_выдача_сим[4];
    define procedure вывсим(in с : сим);
private
   const размер_буф = 32;
   type срс = set of (невозможно at 6, готово at 7);
         счетчик = целый range 0 .. размер_буф;
         индекс= цеший range 1.. размер_буф;
   var хв, хиз : индекс;
        n : счетчик;
       неполон, непуст : сигнал;
       буф : array[индекс] of сим;
   procedure вывсим(in с : сим);
   begin
      if n = размер_буф then
         ждать(неполон)
   end if;
   буф[хв]:=с; хв:=(хв mod размер_буф)+1;
   n:=n + 1;
   послать(непуст)
 end;
```

```
process управ;
var цел at #64 : сигнал;
      статус at #177564 : срс;
      буфрег at #177566 : сим;
 begin
   do  if n = 0 then
          ждать(непуст)
     end if;
        буфрег := буф[хиз]; хиз (хиз := mod размер_буф) + 1;
        статус :=срс(невозможно);
        ждать(цел);
        статус := срс( ); n := n - 1;
        послать(неполон)
     end do
   end;
begin n:=0; хв:=1; хиз:=1;
   init управ
end module буфериз_выдача_сим;
```

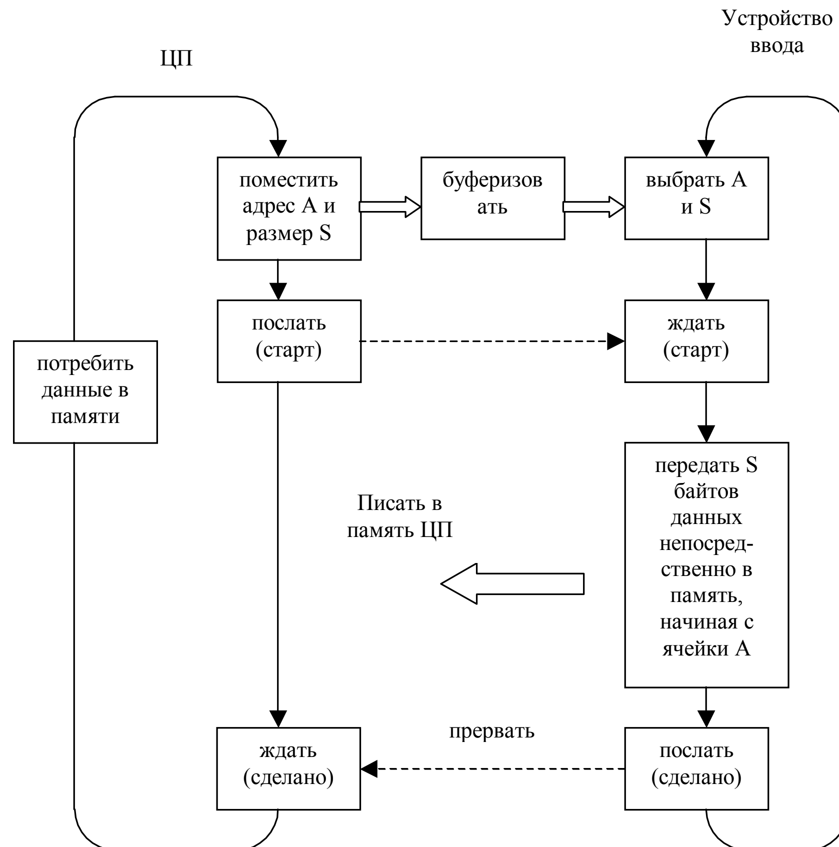
Теперь обращение к процедуре *вывсим* требует расположения символа в буфере, после чего, при условии, что буфер не полон, вызывающий процесс может сразу же продолжить работу. Внутри модуля ввода-вывода управляющий процесс все время разгружает буфер, передавая символы на устройство вывода. Таким образом, вызывающий и управляющий процессы работают в конфигурации поставщик/потребитель, которая была обсуждена в гл. 6. Эффект буферизации состоит в сглаживании асинхронных запросов на распечатку символов. Такой подход можно столь же эффективно использовать для ввода символа и, вообще говоря, для передачи элемента данных любого вида, например строчек распечатывающему устройству, вводных двоичных данных от аналого-цифрового преобразователя и т. д.

7.4.2. Передача блоков данных с использованием МПДП

Передача с использованием МПДП может быть также запрограммирована с помощью модели, аналогичной модели на рис. 7.3; отличие состоит в том, что передаваемые между ЦП и устройством МПДП данные содержат адрес и размер фактического блока данных, подлежащего вводу или выводу, а прерывание сигнализирует о завершении передачи всего блока. Расширенная модель для МПДП имеет, таким образом, вид, изображенный на рис. 7.6. Ее связь с моделью на рис. 7.3 очевидна.

Чтобы иметь возможность программировать устройства МПДП, нужны две системные процедуры низкого уровня. Прежде всего требуется функция *адр*(*x*) для получения адреса переменной *x в* памяти и функция *размер*(*x*) для получения размера переменной *x* в памяти. Эти функции необходимы,

Рис. 7 6. Операция ввода под управлением устройства МПДП.
чтобы установить в контроллер МПДП начальный адрес и переслать значение размера блока.



Чтобы показать, как можно запрограммировать передачи ввода-вывода в системе МПДП с помощью описанных к настоящему моменту языковых конструкций, рассмотрим следующий модуль ввода-вывода, являющийся очень простой управляющей программой для чтения блока символов с диска RK05 машины PDP 11. В примере предполагается, что диск разделен на логические блоки и каждый блок идентифицируется уникальным целым числом *нб*. Предполагается, что функция *отображениедиска*

function *отображение_диска*(*нб*: *целый*): *целый*

отображает значения логических блоков на фактические адреса физического диска. Данный модуль представляет одну про-

цедуру интерфейса с именем *читать_блок* для чтения конкретного блока символов. Работа этой процедуры по существу идентична первой версии процедуры *вывсим* из разд. 7.4.1, т. е. состоит в следующем: подготовить устройство; разрешить прерывания; ждать прерывание; снять прерывание и, наконец, выполнить возврат.

```
device module читать_диск[5];
  define type блок_сим;
     procedure читать_блок(out бс : блок_сим; in нб: целый);
   use function отображение_диска(нб : целый): целый;
 private
   type блок_сим = array[0..511] of сим;
   procedure читать_блок(iot бс : блок_сим; in нб: целый);
     type функц = (писать => 1, читать => 2);
          режим = (стоп => 0, прод => 1);
          сост-диск = record
                        контр at 0 : режим;
                        фнк at 1..3 : функц;
                        невозможно at 6 : логический
                      end;
     var чбус at #177404 : сост_диск; {управление/статус}
         чбсс at #177406 : целый; {счетчик слов}
         чбаб at #177410 : целый; {адрес блока}
         чбад at #177412 : целый; {адрес диска}
     begin
       чбаб := адр(бс); чбсс := размер(бс);
       чбад := отображение_диска(нб);
      чбус := сост_диск(прод, читать, истина);
      ждать(цел);
      чбус.невозможно := ложь
     end;
   begin  end
   module читать_диск;
```

Основной недостаток данного решения состоит в том, что оно слишком ограничено, чтобы быть полезным на практике. Хотя данные будут храниться на диске как последовательность байтов фиксированной длины, интерпретация этих данных пользователем может быть самой произвольной. Например, один пользователь может считать, что блок данных, состоящий из 512 байтов, содержит 512 символов (как в нашем примере); другой же пользователь может рассматривать аналогичный блок на том же самом диске как совокупность 128 чисел в форме с плавающей запятой. В общем случае как размер блока,

так и тип его компонент могут меняться. Из этого следует, что процедура интерфейса *читать_блок* должна разрешать различные типы для своего параметра **out**. В связи с этим неизбежно возникают некоторые серьезные трудности в языке с сильной типизацией, где принадлежность типу определяется статически во время компиляции.

Здесь возможно несколько решений с различным соотношением между гибкостью и надежностью. На одном полюсе находится решение, когда тип всех данных, записываемых на устройство, разбитое на блоки, или читаемых с него, можно специфицировать как вариант одного комбинированного типа. Варианты можно обеспечить с помощью массивов всех предопределенных типов произвольной длины. Это решение надежно, но излишне жестко, так как описание типа записи нужно модифицировать каждый раз при введении нового определенного пользователем типа. На другом полюсе находится решение, когда вводится новый вид типа формальных параметров процедуры, называемой *пространство,* который может употребляться в качестве типа любого объекта. Другими словами, формальный параметр типа *пространство* не реагирует на проверки сильной типизации. С использованием формального параметра типа *пространство* приведенная выше процедура *читать_блок* может быть переопределена следующим образом:

procedure *читать_блок*(**out** *бс* : *пространство*; **in** *нб* : *целый*);

где *бс* представляет теперь переменную любого размера и типа. На практике второе решение является, по-видимому, лучшим. Конечно, оно ненадежно, но если его использование ограничить модулями ввода-вывода, то его влияние на общую надежность сводится к минимуму. Достоинством решения является то, что оно позволяет писать общие процедуры интерфейса, которые могут работать с любыми определенными пользователем типами, и что оно просто в употреблении.

7.5. ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА НИЗКОГО УРОВНЯ В ПРАКТИЧЕСКИХ ЯЗЫКАХ

Как было сказано во введении, основной возможностью, которую предоставляют традиционные языки реального времени в рассматриваемой области, является разрешение помещать в код высокого уровня разделы на языке ассемблера. Такие языки, как CORAL 66 (Вудвард и др., 1970), являющийся языком слабой типизации, позволяют в некоторой степени манипулировать с регистрами устройств, предоставляя битовые операции и возможность адресации определенных ячеек памяти. Однако в общем случае программирование низкого уровня на чисто

последовательном языке не может быть развито достаточно глубоко из-за невозможности выразить параллелизм.

Кратко обрисованные в данной главе конструкции базируются на опыте эксплуатации языка Модула (Янг, 1980). Из представленных здесь конструкций Модула включает лишь небольшое подмножество, но они реализованы очень эффективно (Вирт, 1977с и гл. 12). Кроме Модулы, имеется еще несколько практических языков, предоставляющих аналогичные возможности. Языки «особо надежной» серии и появившийся в итоге язык Ада в своих первоначальных спецификациях проекта содержали требования возможностей программирования низкого уровня, но в конечном счете эти ожидания не были оправданы (Пайл, 1979). Отображение определенных пользователем типов на машинные регистры выполнено адекватно, но общая структура процессов и в особенности обработка прерываний развиты недостаточно. Из «особо надежных» языков язык Yellow (SRI, 1978) характеризуется в этом смысле лучшим проектом, непосредственно следуя Модуле. Между прочим, рассмотренный в разд. 7.4.2 формальный параметр со спецификацией *пространство* берет свое начало в языке Yellow.

Глава 8. ОБРАБОТКА ОШИБОК

8.1. ОСНОВНЫЕ ПОНЯТИЯ

В отличие от коммерческого и научного программного обеспечения программное обеспечение реального времени очень часто должно функционировать непрерывно даже при возникновении ошибочных ситуаций. Например, вычислительная машина, управляющая функционированием большой газовой печи, не может прервать работу печи при возникновении ошибки. Напротив, она должна попытаться продолжить управление работой печи как можно лучше. Возможное при этом понижение качества управления предпочтительнее выполнения последовательности дорогостоящих операций, связанных с остановкой производства. Из этого требования, заключающегося в том, чтобы системы реального времени функционировали при неисправностях, следует, что в языке реального времени должны иметься возможности обработки ошибок.

В проблеме обработки ошибок различаются два аспекта. Во-первых, ошибки должны быть обнаружены; обычно это происходит на двух различных уровнях. На уровне выполнения

ошибки обнаруживаются с помощью объединения средств машинного оборудования и вносимого компилятором проверочного кода. Примерами таких ошибок является деление на нуль и нарушение границ индексов у массивов. На более высоком уровне ошибки можно обнаруживать с помощью явно программируемых проверок вида

if не требуемое условие **then** ошибка ...

С точки зрения языка, труднее устранять последствия ошибок, обнаруживаемых на уровне выполнения, так как они могут встретиться в любом месте программы и в любом месте оператора. Поэтому в момент обнаружения ошибки определить точное состояние вычислительного процесса может оказаться очень трудно или даже невозможно. Конечно, появление обнаруживаемых в процессе выполнения ошибок в принципе можно не допустить, если разрабатывать программу особенно тщательно; но на практике такой подход не реалистичен, и, в частности, на него нельзя полагаться в тех областях применения, когда единственное проявление ошибки может привести к катастрофическим последствиям.

Второй аспект обработки ошибок состоит в необходимости реагировать на обнаруженную ошибку. В обычном случае адекватный ответ может состоять в простой выдаче соответствующей распечатки и последующем завершении выполнения программы. Однако система реального времени должна уметь генерировать более сложные ответы, включающие приведение системы в согласованное состояние, исправление результатов возможных неправильных действий и дальнейшее продолжение нормального функционирования.

Какой бы механизм обработки ошибок ни был введен в язык реального времени, в идеальном случае он должен удовлетворять определенным требованиям. Во-первых, этот механизм должен быть прост для понимания и использования. Во-вторых, обрабатывающий ошибки код не должен быть слишком большим, чтобы не затруднять понимание нормального, свободного от ошибок выполнения программы. В-третьих, механизм обработки ошибок должен быть таким, чтобы соответствующие издержки работы программы проявлялись только лишь при фактической обработке ошибки и не влияли на производительность программы при нормальных условиях. В-четвертых, этот механизм должен позволять единообразную обработку ошибок, обнаруженных как в процессе выполнения, так и программным способом. В-пятых, механизм должен позволять программировать операции восстановления.

В этой главе будут обсуждаться и оцениваться в соответствии с перечисленными выше требованиями конкретные меха-

низмы обработки ошибок. Сначала будут рассмотрены методы, использующие традиционные языковые механизмы, а затем будет представлено современное понятие механизма обработки исключений.

8.2. ТРАДИЦИОННЫЕ МЕТОДЫ ОБРАБОТКИ ОШИБОК

Один из простейших методов обработки обнаруженных программой ошибок состоит в использовании процедурных параметров для возвращения кодов ошибок во внешнюю среду. После обращения к процедуре можно выполнить проверку полученного кода ошибки и обратиться к выполнению соответствующих действий. Основным преимуществом этого подхода является то, что он очень прост и не требует для своего сопровождения никаких новых языковых механизмов. Впрочем, он имеет ряд существенных недостатков, в чем можно убедиться, рассмотрев следующий пример. Предположим, что процедура по имени *задача* обращается к трем процедурам: *ввод_данных, обработка-_данных* и *вывод_данных* следующим образом:

```
procedure задача;
   var a : данные;
 begin
   ввод_данных(d);
   обработка_данных(d);
   вывод_данных(d)
end;
```

Если предполагается, что ошибка может произойти в любой из этих процедур и что при появлении ошибки дальнейшая обработка данных должна быть прекращена и заменена обращением к процедуре *очистить,* то программу *задача* нужно переписать следующим образом:

```
procedure задача;
   var d : данные;
     фош : логический;
begin
     ввод_данных(d, фош);
     if not фош then
       обработка_данных(d, фош);
       if not фош then
        вывод_данных(d, фош)
     end if;
     end if;
     if фош then
        очистить
     end if
  end;
```

где *фош* - флажковый регистр ошибки, который процедура устанавливает при появлении ошибки.

Хотя это решение и работоспособно, оно не может считаться удовлетворительным. Во-первых, введение операции обработка ошибки в форме вложенных операторов **if** существенно усложняет понимание нормального, безошибочного функционирования нашей программы. Во-вторых, обрабатывающий ошибку код, увеличивает время работы программ независимо от того, есть ошибка или нет. Ясно, что этот подход к обработке ошибок не удовлетворяет требованиям, изложенным в разд. 8.1.

В сущности, недостатки этого метода объясняются следующим. Описанные в гл. 4 обычные языковые структуры управления специально построены так, чтобы управление программы было организовано в виде хорошо упорядоченной последовательности. Это сделано для поощрения так называемого структурного программирования. Но обработка ошибки требует, чтобы естественная передача управления в программе была нарушена для анализа чрезвычайной ситуации. Из этого следует; что регистрация ошибки с помощью процедурных параметров с дальнейшим использованием естественно структурированных операторов управления не отвечает требованиям, предъявляемым к механизму обработки ошибок в языке реального времени. К тому же, таким способом не легко обрабатывать ошибки выполнения. Хотя на основе механизма установки глобальных флажков ошибок можно разработать схемы, которые представляли бы процедурам аппаратно задаваемые параметры ошибок или использовали бы какую-либо разновидность программ обработки аварийных ситуаций, все они оказываются слишком сложными и значительно увеличивают время работы программы.

Более аккуратное решение проблемы обработки ошибок можно получить, если предоставить возможность обходиться без обычных программных механизмов управления. Одним из очевидных способов сделать это является введение в язык оператора **goto.** Если при обычной разработке программ неограниченное использование конструкций типа **goto** безусловно приносит вред, то их использование при обработке ошибок может упростить разработку программы и сделать ее более ясной. В случае рассматриваемого нами примера вызываемые программой *задача* процедуры вместо установки флажка ошибки могут непосредственно передавать управление к метке ошибки внутри программы *задача,* т. е.

```
procedure задача;
  var d : данные;
  label плохо;
procedure ввод_данных(out d : данные);
  …
  if ошибка then goto плохо end if;
end;
procedure обработка_данных(inout d : данные);
    …
  if ошибка then goto плохо end if;
    …
  end;
     и т. д.
```

```
begin
  ввод_данных(a);
  обработка_данных(d);
  вывод_данных(d);
  return;
  плохо: очистить
end;
```

Теперь структура процедуры *задача* гораздо прозрачнее по сравнению с предыдущей версией, а время работы программы увеличивается только тогда, когда ошибка имеется на самом деле. Следует подчеркнуть, что если оператор **goto** используется таким способом, то это означает не только операцию машинного перехода, но также и ненормальное завершение процедуры. Поэтому состояние стека должно «свертываться» до тех пор, пока среда не вернется в состояние процедуры, в которую было передано управление, для нашего случая в состояние *задача.* В общем случае передача управления может произойти из процедуры, выполняемой в результате последовательности вложенных обращений на любую глубину, в любую из активных процедур в цепочке обращений. Свертывание стека в таких случаях может потребовать определенного времени, но обычно это приемлемо, так как эти расходы происходят только при обработке ошибки, а не во время нормальной работы программы.

Гибкость операторов **goto** может быть значительно увеличена за счет применения переменных меток. В приведенном выше примере метка *плохо* является постоянной меткой и не может быть изменена во время работы программы. Однако часто бывает так, что ответ на заданное условие ошибки должен быть различен в разных контекстах; это, в частности, справедливо для ошибок, обнаруживаемых машинным оборудованием. Предположим, например, что всегда при обнаружении арифметической ошибки выполняется оператор

```
goto мош
```

где *мош* — глобальная меточная переменная, заявленная в описании

var *мош*: *метка*;

Рассмотрим теперь следующий эскиз программы:

```
program использование_пер_меток;
   var мош : метка;
   procedure a;
         …
   end;
   procedure b;
      var стар_мош: метка;
   begin
      стар_мош := мош; мош := вплохо; {загрузить
                                         новую мош}
            …
      a;
            …
      goto выход;
  вплохо : {обработка арифметических ошибок}
  выход : мош := стар_мош {восстановить мош}  end;
  begin
  мош := плохо;
      …
    a; b;
     …
  плохо: {обработка арифметических ошибок}
  end
```

Главная программа устанавливает метку ошибки *мош* в состояние *плохо,* так что все ошибки, встречающиеся внутри главной программы или внутри процедуры а при ее непосредственном вызове, обрабатываются кодом с меткой *плохо.* Но процедура *b* содержит свою собственную подпрограмму обработки арифметических ошибок. При входе в *b* значение *мош* записывается в локальную меточную переменную, а ее новым значением становится значение *вплохо.* При выходе из процедуры *b* восстанавливается старое значение *мош.* Следовательно, каждая из ошибок, встречающихся внутри процедуры *b* или внутри процедуры *a* при ее вызове процедурой *b,* будет обрабатываться подпрограммой *вплохо.*

Этот пример интересен тем, что он показывает исключительную гибкость аппарата меточных переменных. При возникновении ошибки в а соответствующая программа обработки ошибки определяется автоматически во время работы программы по

контексту, в котором произошло обращение к *a.* Возможность динамической связи ошибки с одной из нескольких подпрограмм обработки этой ошибки особенно важна в тех случаях, когда нужно обеспечить операции восстановления после ошибки.

Из приведенного выше краткого описания метода использования операторов **goto** и переменных меток можно увидеть, что он удовлетворяет, по крайней мере в некоторых отношениях, всем перечисленным в разд. 8.1 требованиям. Основной недостаток этого подхода заключается в его очевидной ненадежности; если мы хотим избежать ошибок при программировании, то должны быть очень внимательны, особенно при манипулировании меточными переменными. Кроме этого, может оказаться очень трудно понять поведение программы в случае обнаружения ошибки, исследуя только текст программы. Таким образом, признавая операционную адекватность аппарата передач управления и меток, нужно отметить, что она обладает чертами ненадежности и трудности понимания, присущими средствам низкого уровня. Как следствие этого, для обработки ошибок были разработаны средства высокого уровня, к обсуждению которых мы и переходим.

8.3. ОБРАБОТКА ИСКЛЮЧЕНИЙ

8.3.1. Структурная обработка ошибок

Как было отмечено в конце предыдущего раздела, простой оператор **goto** и переменные метки представляют эффективный механизм обработки ошибок. Каждая конкретная ошибка может обрабатываться различными способами в зависимости от динамического контекста, в котором она происходит. Но, к сожалению, этот механизм является механизмом слишком низкого уровня для общего использования в языках реального времени. Поэтому наряду с желательностью заменить операторы **goto** структурными операторами управления, такими, как **if, case, while** и т. д. при разработке обычных программ, хотелось бы заменить их структурным механизмом и при проектировании программ обработки ошибок. Прежде чем описать, как такой механизм может быть реализован, сначала необходимо рассмотреть, какие последствия влечет за собой структурная обработка ошибок.

Предположим, что при выполнении вычислений может произойти ошибка *E*. Тогда для обработки ошибки *E* можно определить несколько подпрограмм обработки ошибки. Сейчас нам не обязательно знать точно, что делает такая подпрограмма. Достаточно предположить, что при своем возбуждении

она генерирует адекватный ответ на появление ошибки. С каждой такой подпрограммой связывается некоторая область, которая специфицирует тот интервал вычислений, для которого появление ошибки *E* означает активацию этой подпрограммы. Заметьте, что вместо слова «программа» мы используем слово «вычисление». Мы прибегаем к этому для того, чтобы подчеркнуть, что ошибки встречаются и должны обрабатываться во время исполнения программы. Таким образом, области являются динамическим понятием, достаточно свободно связанным со статическим текстом программы.

В сущности, структура систем обработки ошибок зависит в первую очередь от этого понятия области ошибки. Из структурной обработки ошибок вытекает необходимость идентификации многочисленных различных ответов, которые должны быть выданы вслед за обнаружением конкретной ошибки (в зависимости от того, где произошла ошибка), и в дальнейшей фиксации каждого интервала, в котором требуется один и тот же ответ, в качестве области. Затем для каждой такой области специфицируется соответствующая подпрограмма обработки ошибки.

Точность, с которой можно специфицировать область, определяет, насколько точно можно определить положение ошибки и насколько аккуратно можно ее обработать. Так как размер области соответствует подразделу вычисления, который мы обрабатываем при исполнении программы, то очевидно, что области должны специфицироваться в терминах выполняемых программных единиц, т. е. блоков, операторов управления или простых операторов. Из них наиболее подходящей единицей является блок (т. е. простой блок, процедура или функция), так как это позволяет варьировать точность спецификации области. Например, если ошибка должна локализоваться с точностью до простого оператора, то можно описать простой блок, содержащий только один оператор. Но, как правило, ошибки должны локализироваться с точностью до группы операторов, представляющих отдельное логическое действие, а они, естественно, представляются процедурами или функциями.

Взяв в качестве основной единицы областей блок, можно приступить к спецификации размера области. Связанная с блоком *B* область ошибок включает все операторы, которые будут выполняться при выполнении *B,* но исключает все те операторы, которые лежат внутри областей, связанных с вызываемыми *B* блоками.

Чтобы лучше понять смысл данного определения, рассмотрим следующий эскиз программы, для которой определяются

две области; одна — связанная с процедурой *P*1*,* а другая — с процедурой *P*2.

```
procedure P1;
  procedure P3;
  begin
     …;
     …;
  end;
  procedure P2;
    begin [область 2]
      …;
       P3;
      …;
    end;
begin [область 1]
 …;
 …;
 P3;
 …;
  P2;
 …
end;
```

В данном примере любая встречающаяся в *P*1 ошибка находится в области 1; аналогично, любая ошибка, встретившаяся в *P*2, находится в области 2. Но ошибка, встретившаяся в *P*3, находится в области 1, когда *P*3 вызывается непосредственно из *P*1, и находится в области 2, когда *P*3 вызывается из *P*2. В этом примере подчеркивается динамическая природа областей (отметьте также аналогию между этим примером и программой *использованив_пер_меток* из разд. 8.2). Исполнение процедуры *P*1 во времени изображено графически на рис. 8.1, на котором легко увидеть, что область 2 вложена в область 1. С точки зрения разработки программы это соответствует тому, что имеется локальная область в процессе вычисления, обозначенном *P*1, в которой ошибки должны обрабатываться специальным образом. Изложенное выше абстрактное понятие области ошибки мы теперь сформулируем в более конкретных терминах, описав особые конструкции, которые позволят реализовать этот структурный подход в языке реального времени. В соответствии с традицией для обозначения факта появления ошибки будет использоваться термин *исключение,* а соответствующая подпрограмма обработки будет называться *подпрограммой обработки исключений.* В следующем разделе будет изложена базовая форма механизма обработки исключений, и затем в разд. 8.3.3 будет показано, как можно сформулировать правила распро-

странения исключений, что позволит абстрактное понятие области ошибки реализовать на практике. В дальнейших разделах рассматриваются проблемы использования исключений для восстановления программы после появления ошибок, приме-

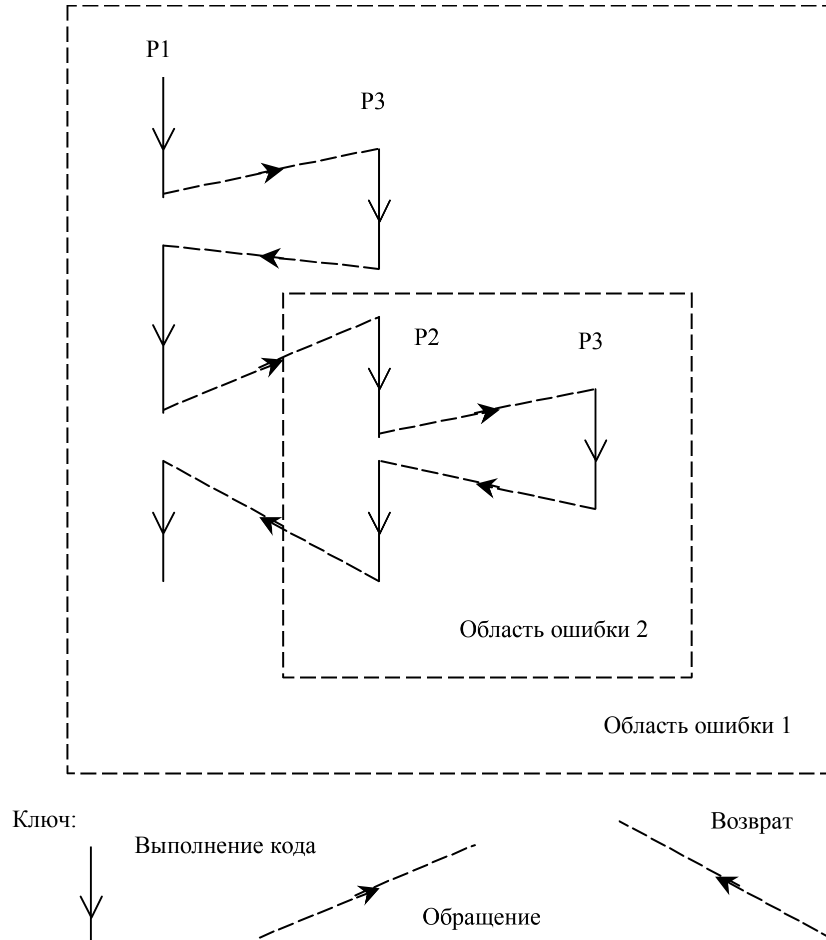


Рис. 8.1. Выполнение процедуры *P1* и соответствующие области ошибок.

нения исключений в мультипроцессной среде и реализации подпрограмм обработки исключений.

8.3.2. Базовые механизмы

В данном разделе будет разработана базовая схема для представления исключений и подпрограмм их обработки. Для простоты мы сразу же предположим, что процедуры не производят вложенные обращения к другим процедурам, так что областью подпрограммы обработки ошибок является просто тело процедуры, с которой связана эта подпрограмма.

Первое требование состоит в том, чтобы обеспечить средство спецификации имен всех исключений, которые могут произойти в программе. Наиболее удобным способом сделать это является введение абстрактного типа с именем *исключение,* так что-

бы объекты этого типа можно было описывать при необходимости в любом месте. Так, например,

var *ex*1*, ex*2 : *исключение*;

описывает два исключения с именами *ex*1 и *ex*2*.* Исключение может находиться в двух состояниях: невозбужденном и возбужденном. В возбужденное состояние оно переводится для того, чтобы указать на появление ошибки, и это может произойти двумя способами. Во-первых, оно может быть возбуждено явно с помощью оператора исключения вида

raise *ex*1

Отметим, что другой возможный способ представления операции возбуждения с помощью обращения к стандартной процедуре, например *raise*(*ex*1)*,* не годится, так как смысл возбуждения исключения состоит в нарушении нормального выполнения программы, в чем мы убедимся позднее. Использование обращения к процедуре предполагает, что управление в конечном счете будет передано обратно, а это может быть не всегда (или вообще никогда) верно. Во-вторых, исключение может быть возбуждено неявно при выполнении программы. В последнем случае имена возбуждаемых таким образом исключений должны быть предопределены. Эти два метода возбуждения исключения соответствуют двум типам ошибок: программно-обнаруженным и обнаруженным при выполнении. Независимо от того, как возбуждено исключение, оно обрабатывается соответствующей подпрограммой обработки исключений одним и тем же способом. Подпрограммы обработки исключений удобно располагать сразу же за основным телом блока, с которым они связаны. Так как с одним блоком может быть связано несколько подпрограмм обработки, удобной системой обозначений является модифицированная форма оператора **case**. Например, следующая процедура *E*1 содержит описания для исключений *ex*1 и *ex*2*,* а также подпрограммы их обработки.

```
procedure E1;
 var ex1, ex2 : исключение;
 begin
   …
 except
   ex1 : (S1);
   ex2 : (S2)
  end;
```

Здесь *S*1 обозначает последовательность операторов, которые нужно выполнить при возбуждении в *E*1 исключения *ex*1*,* и аналогично *S*2 обозначает последовательность операторов,

которые нужно выполнить при возбуждении в $E1$ исключения *ex*2.

Иногда бывает удобно предоставить одну подпрограмму обработки нескольким исключениям. Это можно сделать, как и в случае оператора **case,** разрешив именам нескольких исключений помечать одну и ту же подпрограмму, например,

```
        except
            ex1, ex2, ex3: (Sa);
            ex4, ex5 : (Sb)
              ... и т. д.
    end;
```

И наконец, заметим, что все использования имени исключения, т. е. в операторах **raise** и подпрограммах обработки исключений, должны находиться в области действия [9] описания исключения. Это необходимо как с точки зрения согласованности с другими типами данных, так и для обеспечения возможности достаточно эффективной реализации. Это, однако, не означает, что область подпрограммы обработки исключений не может лежать вне границ статической области действия. Этот последний момент обсуждается подробнее в следующем разделе.

Выше мы наметили удобную синтаксическую схему для описания исключений и определения подпрограмм обработки исключений. Впрочем, до сих пор мы совершенно не рассматривали вопрос, что фактически делает подпрограмма обработки исключений. Возвращаясь к приведенному выше примеру процедуры $E1$, предположим, что в теле $E1$ возбуждено исключение *ex*l. В этом случае возможны три типа действий:

(a) Выполнение всех оставшихся операторов из $E1$, следующих за точкой, в которой произошло исключение, прекращается; передача управления осуществляется непосредственно на подпрограмму обработки исключения, которая затем и выполняется. После завершения работы подпрограммы процедура заканчивает свою работу обычным способом, возвращая управление вызывающей среде.

(b) Управление передается сразу же подпрограмме обработки ошибок, которая затем выполняется. При завершении подпрограммы обработки управление возвращается оператору, непосредственно следующему за оператором, в котором было возбуждено исключение.

(c) Управление передается сразу же подпрограмме обработки ошибок, которая затем выполняется. Внутри подпрограм-

[9] В этом тексте английский термин domain переводится словом область, а термин scope словами область действия. — *Прим ред.*

мы может быть задействован оператор **resume**, в этом случае управление возвращается прерванной процедуре, как в случае (b); если завершение подпрограммы происходит без использования оператора resume, то процедура прекращается, как в (a).
В каждом из этих трех случаев вызывающее активацию подпрограммы обработки ошибок исключение автоматически перестает быть возбужденным непосредственно перед входом в подпрограмму.

В очень обобщенном анализе обработки исключений Гудинаф (1975) предложил систему, в которой можно воспользоваться любой из этих трех возможностей. Он назвал их исключениями *уйти, отметить* и *разобраться* соответственно. К сожалению, в его анализе обработка исключений рассматривалась как общий метод программирования, применимый к очень широкому классу проблем. Мы же рассматриваем обработку исключений лишь как средство обработки ошибок в системах реального времени. В этом контексте необходимо учесть следующие три фактора:

(i) Механизм должен отвечать требованиям программирования операций восстановления после ошибок.

(ii) Использование этого механизма не должно мешать получению ясных и надежных программ.

(iii) Этот механизм должен допускать такую реализацию, при которой время работы программы увеличивалось бы только при фактическом возбуждении исключения.

Механизм исключения *уйти* (a) подразумевает, что исключения представляют всегда условия завершения. Поэтому на первый взгляд этот подход не кажется пригодным для программирования действий по восстановлению после ошибки. С другой стороны, механизм *отметить* (b) кажется идеальным. Подпрограмма обработки ошибок может исправить состояние вычислительного процесса в той точке, где было возбуждено исключение, а затем возобновить вычисление так, как если бы до этого ничего не произошло. Но на самом деле у механизма *отметить* имеются серьезные недостатки. Во-первых, может оказаться, что возбужденные реализацией исключения с трудом поддаются исправлению. Например, арифметическое переполнение, возникшее в середине последовательности сложных выражений, может привести к ситуации, когда к подпрограмме обработки ошибок обращаются с несколькими частичными результатами, хранящимися в различных машинных регистрах. Этого можно избежать, только запретив компилятору выполнять какую бы то ни было оптимизацию с использованием машинных регистров; каждое вычисление он должен делать в строго пред-

писанном порядке, немедленно возвращая каждый вычисленный результат обратно в память. Поэтому обеспечение только исключений *отметить* неизбежно ухудшает производительность программы. Во-вторых, исключения *отметить* очень сильно затемняют программы. Рассмотрим, например, следующую последовательность операторов:

сум := $a + b$;
x := *сум* - a;
{утверждается, что здесь $x = b$}

Если только при вычислении $a + b$ не произойдет переполнения, приведенное в комментариях утверждение всегда будет истинным. Однако если переполнение произойдет, то программа обработки для исключений *отметить* может при восстановлении заслать в *сум* любое значение, какое она выберет, и затем возвратиться к следующему оператору с таким результатом, с которым невозможно доказать требуемое утверждение. Исключение *уйти* не страдает ни от одного из этих недостатков. Компилятор свободен в выборе оптимизации, так как при возбуждении исключения все последующие операции прекращаются. Программы становятся более ясными, и их правильность легче доказывать, так как подпрограмма обработки не вводит дополнительного прерывания в естественный поток вычисления, а является скорее заменой остатка укороченной процедуры. Наконец, действия по восстановлению после ошибки в случае механизма *уйти* можно запрограммировать, либо разработав подпрограмму обработки ошибок так, чтобы она перевычисляла всю прекращенную процедуру после исправления входных параметров, либо построив такую подпрограмму обработки, которая бы ликвидировала причину ошибки и затем заново вызывала прекращенную процедуру. Конечно, оба этих метода замедляют работу программы по сравнению с исправлением ошибки с использованием исключения *отметить,* так как в обоих случаях вычисление повторяется заново с самого начала, а не возобновляется с середины после короткой паузы. Тем не менее это не противоречит требованиям восстановления после ошибки, которые были установлены ранее и состоят в том, чтобы все издержки механизма *уйти* во время работы программы проявлялись только при фактическом проявлении ошибки.

С учетом всех факторов, которые мы только что обсудили, становится очевидным, что операция *уйти* является единственно приемлемым механизмом для обработки ошибок в системе реального времени. Оба других исключения *отметить* и *разобраться* должны быть отвергнуты; второе — по той причине, что оно имеет все недостатки исключения *отметить* и что

программы становятся еще менее понятными в связи с дополнительной неопределенностью относительно возобновления (или прекращения) прерванной процедуры.

8.3.3. Распространение исключений

В разд. 8.3.1 было введено понятие области ошибок. В данном разделе будет показано, как такие области можно реализовать на практике, разрешая исключениям переходить из прерванной процедуры в вызывающую среду.

Когда внутри процедуры возбуждается исключение, то в случае, если подпрограмма обработки ошибок описана локально в этой процедуре, происходит ее инициация. Однако если внутри этой процедуры не существует подпрограммы обработки данного исключения, то процедура завершается, а управление возвращается вызывающей процедуре, в которой немедленно перевозбуждается то же самое исключение. Если в этой процедуре существует подпрограмма обработки данного исключения, то исключение обрабатывается этой подпрограммой; в противном случае описанный процесс повторяется снова. Процедура завершается, а исключение перевозбуждается в вызывающей процедуре. Эта операция повторяется до тех пор, пока не будет найдена подпрограмма обработки для передаваемого исключения. Это правило распространения исключений можно считать операционным определением области подпрограммы обработки ошибок.

Чтобы показать, что это правило эквивалентно абстрактному определению области, рассмотрим снова выполнение процедуры $P1$, показанной на рис. 8.1. В терминах подпрограмм обработки исключений область 1 будет обозначаться включением подпрограммы обработки исключений в тело процедуры $P1$, а область 2 будет обозначаться включением подпрограммы исключений в тело процедуры $P2$. Теперь при возникновении исключения в $P2$ оно будет обрабатываться локальной подпрограммой обработки исключений в $P2$. Но когда исключение возникает в $P3$, то, так как в $P3$ не содержится никакой подпрограммы обработки исключений, эта процедура завершается, а исключение перевозбуждается в вызывающей процедуре. Таким образом, если $P3$ вызывается непосредственно процедурой $P1$, то исключение перевозбуждается и обрабатывается в $P1$; если же $P3$ вызывается процедурой $P2$, то исключение перевозбуждается и обрабатывается в $P2$. Отсюда видно, что обозначенные на рисунке абстрактные области реализуются с помощью правила распространения исключений.

В связи с применением этого механизма возникает одна небольшая трудность, которая состоит в том, что исключение может быть передано за границы области его описания. Это естественно следует из понятия области ошибки, обозначающей скорее интервал вычислений, чем интервал программного текста. Простым и естественным решением этой проблемы является введение исключения *вне_области,* так как оно в конечном счете неизбежно должно привести к возврату а первоначальную область. Например, рассмотрим следующую ситуацию:

```
procedure вне_области; forward;
module x;
   define procedure y : procedure z;
   use procedure вне_области;
private
   var e: исключение;
   procedure y;
   begin
     …
     if ... then raise e endif;
      …
   end;
   procedure z;
   begin
      …;
   вне_области;
      …;
   except
     e : (подпрограмма обработки исключений для e)
   end;
begin
end module x;
procedure вне_области;
begin
  …
   y;
end;
```

Обращение к процедуре *z* в модуле *x* приведет к вызову процедуры *внв_области,* которая в свою очередь обратится к процедуре *у.* Если исключение *е* возбуждено в *у,* го оно передается в процедуру *внв_области* и затем в *z,* где оно, наконец, и обрабатывается. Хотя в конечном счете это исключение и обрабатывается, отсутствие возможности реагировать на исключение в процедуре *внв_области* может иногда приводить к трудностям. Например, предположим, что процедура *вне_области* открыла

файл, который недоступен для подпрограммы обработки исключений в *z*. Если процедура *вне_области* прекращается преждевременно, то этот файл останется открытым, возможно, в несогласованном состоянии. Ясно, что необходим механизм, с помощью которого процедура *вне_области* могла бы выполнить некоторые действия по согласованию состояний, прежде чем передавать исключение дальше. Это можно сделать, если ввести специальное зарезервированное имя исключения *любое,* которое обозначает любое исключение, а также ввести специальный вид оператора **raise**, в котором не задается никакого имени исключения, т. е.

raise;

выполнение этого оператора приводит к перевозбуждению последнего возбужденного исключения. Имея это минимальное расширение, каждую процедуру, аналогичную рассмотренной выше процедуре *вне_области,* можно построить так, чтобы она завершалась вполне аккуратно, обрабатывая исключение локально с помощью подпрограммы исключений *любое* и передавая исключение с помощью анонимного оператора raise. Такой вид работы называется программированием последних целей; подробнее он описывается в следующем разделе.

И наконец, исключение может передаваться за границы области постоянно двумя способами. Во-первых, можно описать исключение и не задать для него никакой подпрограммы исключений. Вообще говоря, это может считаться программистской ошибкой и в принципе может быть обнаружено компилятором. Кроме этого, обычно вводится некоторая внешняя «защитная сетка» в виде помеченной меткой любой подпрограммы исключений, которая автоматически вносится реализующей системой в самую внешнюю область действия в программе. Во-вторых, если язык позволяет использовать глобальные операторы **goto**, то структура областей ошибок может подвергаться самым невероятным преобразованиям. Впрочем, нет серьезных причин для введения таких операторов в язык, в котором имеется возможность обработки исключений, и поэтому эта трудность легко обходится.

8.3.4. Восстановление после ошибок

Наиболее широко подпрограммы обработки исключений будут использоваться для обеспечения традиционных возможностей сообщения об ошибках, когда неудавшаяся операция не вырабатывает никаких полезных результатов и исключение возбуждается лишь для того, чтобы сообщить сам этот факт. Такое использование подпрограмм обработки исключений можно

сравнить с использованием процедурных параметров ошибок, описанных в разд. 8.2. Но одним из требований к механизму обработки ошибок в языке реального времени является то, что он должен предоставить возможность для программирования восстановления после ошибок. В этом разделе будут в общих чертах описаны несколько методов обеспечения этой возможности с помощью описанного механизма обработки исключений, который и в этом отношении обладает определенными преимуществами.

Простейшим способом восстановления после ошибки, случившейся в процедуре или функции, является использование подпрограммы исключений для выработки корректных значений результатов при сбое вычислительного процесса. Например, следующая функция вычисляет произведение двух целых чисел:

```
function умножить(x, y : целый): целый;
begin
   return x*y
except
   чис_ош : (if знак(x) = знак(y) then
                return максцел
             else
                return - максцел
             endif)
end;
```

Здесь произойдет переполнение, когда произведение превысит максимальное значение из диапазона целого типа. В этом случае предполагается, что автоматически будет возбуждено предопределенное исключение *чис_ош,* в результате чего подпрограмма исключения возвратит максимальное или минимальное целое число в зависимости от ситуации.

Разумеется, приведенный выше пример достаточно тривиален. В случае более сложных вычислений может оказаться необходимым гарантировать осмысленность и пригодность результата даже тогда, когда ошибка встретилась в вычислении. Этого можно добиться, если так построить подпрограмму исключений, чтобы она вычисляла требуемый результат с помощью другого, более простого алгоритма. Как правило, это будет означать, что точность возвращаемых подпрограммой результатов будет хуже точности результатов, выдаваемых при нормальном исполнении главной процедурой. Таким образом, мы имеем возможность существенно ослабить спецификации процедуры в обмен на более надежное функционирование.

Чтобы проиллюстрировать этот метод, предположим, что нужна процедура, у которой входными данными является массив из *n* чисел, представляющих фиксированные значения на

сегменте для некоторой неизвестной непрерывной функции. Процедура должна оценить экстремальное значение этой функции с наибольшей возможной точностью. Основное тело такой процедуры может в общем случае состоять из сложной программы полиномиальной аппроксимации. Однако при возникновении ошибки процедуре разрешается вернуть вместо искомого значения его примерную оценку. Приемлемым эскизом такой процедуры могла бы быть следующая программа:

```
procedure оценить_пик(in x : вектор;
                          out пик: плавающий);
begin
  вычислить экстремальное значение x, используя
  метод полиномиальной аппроксимации
except
  любой: (block
            var i: целый range 1.. n;
          begin
            пик := 0.0;
            for i in 1..n do
              if пик < x[i] then
                пик :—x[i]
              end if
            end do
          end)
 end;
```

Здесь подпрограмма обработки ошибок оценивает экстремальное значение очень грубо, находя максимальное значение вектора *x*. Выполняемый подпрограммой исключений алгоритм настолько прост (и менее точен), что вероятность появления ошибок во время его выполнения очень мала. Следует отметить, что этот метод восстановления после ошибок тесно связан с понятием блоков восстановления, которое было разработано для построения устойчивых по отношению к ошибкам вычислительных систем (Рандел, 1975).

Наряду с попытками обеспечить, чтобы процедура или функция каждый раз возвращала полезные результаты, восстановление после ошибок предполагает также ликвидацию всех вредных последствий, возникших в результате появления ошибки, прежде чем эти последствия начнут сказываться на других частях системы. В этом контексте большой интерес представляет идея уровневого восстановления, когда каждое подвычисление в системе отвечает на возбуждение исключения прежде всего тем, что выполняет собственные локальные операции восстановления, передавая затем исключение обратно к

вызывающей среде. Другими словами, каждой процедуре в последовательности обращений, ведущих к ошибкам, разрешается выразить свою последнюю цель прежде, чем она будет завершена исключительным образом. В качестве примера рассмотрим следующий эскиз программы, в которой процедура с именем *анализ* использует две процедуры *обработка_файла* и *вычисление* для анализа содержимого файла данных.

```
procedure анализ(in f : тип_файла;
                 out результаты : данные);
   var e: исключение;
   procedure вычисление(in x : данные; out y : данные);
   begin
     …
      if ... then raise e endif;
     …
   end;
   procedure обработка_файла(inout f: тип_файла;
                              out y : данные);
      var x : данные;
   begin
      открыть _файл(f);
      читать _данные(f, x);
       вычислить(x, y);
      закрыть_файл(f)
    except  любое : (закрыть_файл(f);
                     raise)
    end;
 begin
    обработать_файл(f, результаты);
 except
    e : ({подпрограмма обработки исключения e})
 end;
```

Когда исключение *е* возбуждается процедурой *вычисление,* процедура *обработка_файла* обычно будет сразу же прекращаться, а управление будет возвращено подпрограмме обработки для *е* в процедуре *анализ.* Это приведет к тому, что файл *f* останется открытым. Чтобы избежать этого, процедура *обработка_файла* построена так, что она перехватывает каждое исключение, которое появляется в процедуре *вычисление,* для того чтобы закрыть открытый файл *f* перед тем, как перевозбудить это исключение в вызывающей среде. Таким образом для подпрограммы обработки исключений обеспечивается, что все под-

вычисления, выполняемые при вложенных обращениях, остаются в согласованном состоянии.

Приведенные выше примеры показывают некоторые из возможностей подпрограмм обработки исключений при обработке ошибок и восстановлении после них. Ответ на вопрос, является или нет этот механизм адекватным, носит достаточно субъективный характер. Ясно, что могут возникнуть такие ошибочные ситуации, в которых подпрограммы исключений не смогут ничего сделать. Например, процедура может выбрать последовательность ячеек из свободной памяти и оставить всю систему в испорченном состоянии. Чтобы работать с ошибками такого типа, необходимо предоставить гораздо более сложные возможности, позволяющие запоминать состояние всей системы в ключевых точках вычисления, что, например, делается в упомянутой ранее блочной системе восстановления. Однако следует запомнить ту важную особенность, что описанный здесь механизм обработки исключений не требует никакой специальной поддержки со стороны машинного оборудования и увеличивает время работы программы только в тех случаях, когда исключение произошло фактически. Поэтому он представляет хороший компромисс между универсальностью и эффективностью. Кроме этого, следует также запомнить, что подпрограммы обработки исключений наиболее часто используются тогда, когда они являются частью проекта очень надежного языка, так что возможность ошибок программного обеспечения, которые могут привести к необратимым повреждениям системы, по своей природе минимальна.

8.3.5. Исключения в параллельных процессах

До сих пор обсуждение обработки ошибок ограничивалось случаем только последовательных программ; но, как отмечено в гл. 6, системы реального времени очень часто описываются не с помощью одной последовательной программы, а с помощью множества взаимодействующих параллельных процессов. Поэтому необходимо рассмотреть, как должен работать механизм исключений в мультипроцессной среде.

Первое, что нужно подчеркнуть, — это то, что в последовательной программе исключения можно рассматривать как синхронные прерывания. Вследствие этого поведение программы при возникновении исключения легко определяется. Однако исключение, возбужденное в процессе в результате действий некоторого другого процесса, будет, как правило, асинхронным событием. В этом случае может оказаться очень трудным определить поведение программы, так как нет никаких гарантий, что получающий исключение процесс будет в состоянии его обрабатывать. Поэтому очень важно как с точки зрения ясности

программы, так и с точки зрения эффективности ее реализации строго ограничить допустимые типы взаимодействий процессов в случае возникновения исключений.

Имеются две основные ситуации, в которых должна существовать возможность возбудить исключение в некотором другом процессе. Во-первых, после того как про процесс стало точно известно, что он функционирует неверно, может оказаться необходимым его прекратить. В таких_случаях прямое использование процедуры прекращения не всегда может оказаться удобным, так как тогда у прекращаемого процесса не остается возможности выполнить перед завершением какие-либо операции согласования. Для решения возникающих в связи с этой ситуацией задач можно ввести предопределенное исключение с именем *отключить*. Исключение *отключить* может быть возбуждено в любом процессе *P* с помощью некоторого процесса, выполняющего оператор отключения вида **fail** *P*. Выполнение оператора **fail** никак не влияет на работу выполняющего его процесса, если, конечно, он сам не отключается. В процессе *P* исключение *отключить* обрабатывается обычным образом. Процесс может выполнить любые требуемые операции завершения по исключению и затем аккуратно закончить свою работу.

Вторая ситуация, в которой может потребоваться передавать исключения между процессами, возникает тогда, когда два процесса взаимодействуют и один из этих процессов отключается, прекращается или попадает в какой-либо другой вид ошибочного состояния. В таких случаях может оказаться необходимым предупредить второй процесс о том, что его партнер начинает функционировать ненормально. Это можно сделать, используя второе предопределенное исключение *ош_связь*.

То, как исключение *ош_связь* будет использоваться, зависит от того, каким образом обеспечивается связь между процессами. В гл. 6 мы описали два основных механизма: монитор и рандеву. Поэтому возможные использования исключения *ош_связь* продемонстрируем, кратко рассмотрев функционирование двух этих механизмов в ситуации, когда появляются ошибки.

Механизм монитора дает возможность процессам устанавливать связь друг с другом, выполняя интерфейсные процедуры в режиме взаимного исключения. Это позволяет процессам также синхронизироваться в мониторе, обмениваясь сигналами. При обращении процесса к монитору он может находиться в одном из трех состояний: ожидая входа в монитор, активно выполняя мониторную процедуру или ожидая в мониторе, когда ему будет послан сигнал. Если процесс отключается или прекращается в то время, когда он ожидает входа в монитор, тогда все, что необходимо сделать,— это отменить запрос на обращение, удалив его из очереди обращений к мониторам.

Здесь не возникает вопроса о возбуждении исключения *ош_связь* в других «клиентах» монитора, так как ясно, что никакого нарушения структур данных монитора произойти не могло. Однако если процесс вынужден покинуть монитор ненормальным образом из-за возбуждения исключения или вызова процедуры прекращения, то, безусловно, существует возможность, что монитор будет оставлен в несогласованном состоянии. При таком положении дел все имеющиеся и возможные клиенты монитора должны быть предупреждены об этом с помощью исключения *ош_связь*. Это можно сделать, отменив все обращения к процессам, которые ждут входа в монитор, и принудив все процессы, которые фактически находятся в мониторе, ожидая сигналов, выйти из монитора немедленно. Тогда в каждом таком процессе в момент обращения к монитору возбуждается исключение *ош_связь*. Любые последующие обращения к монитору инициируют немедленное возбуждение исключения

о*ш_связь*.

Конечно, не всегда может быть верно, что прекращение процесса в мониторной процедуре приведет к такому состоянию монитора, что дальше пользоваться им будет нельзя. В связи с этим необходимо ввести компиляторную директиву, которая позволяла бы подавлять описанный механизм о*ш_связь*. Заметим также, что там, где мы вынуждаем процесс покинуть монитор, используя исключение *отключить,* в монитор может быть введена подпрограмма исключения для исправления всех неправильностей перед перевозбуждением исключения в вызывающем процессе. С помощью этого аппарата о*ш_связь* может передаваться клиенту монитора только тогда, когда это действительно необходимо.

Механизм рандеву представляет несколько отличный метод межпроцессной связи, состоящий в том, что два процесса могут установить связь только при синхронизации друг с другом. Механизм же монитора в этом отношении представляет значительно более свободные возможности налаживания между процессами двусторонней связи. В связи с такой «интимной» природой рандеву использование о*ш_связь* при возникновении ошибки необходимо только для тех процессов, которые непосредственно взаимодействуют друг с другом.

Точка, в которой два взаимодействующих процесса фактически синхронизируются, задается в процессе-служащем оператором **accept**. Любое нормальное исключение, возбужденное в этом операторе, может рассматриваться как принадлежащее обоим процессам. Поэтому оно может обрабатываться локально в рамках оператора **accept** или, если соответствующей подпрограммы обработки нет, передаваться обратно обоим процессам. Основная трудность при использовании механизма рандеву

возникает тогда, когда выполнение одного из процессов прекращается (либо при обращении к процедуре *прекратить,* либо при возбуждении исключения *отключить*). В этом случае выбор соответствующего действия зависит от того, какой процесс был прекращен и какого этапа развития достигло рандеву. Ясно, что если прекращен процесс-служащий, содержащий оператор приема, то единственным возможным действием является завершение рандеву, так как его продолжение не имеет смысла. Затем можно предупредить вызывающий процесс, что рандеву оказалось неудачным, возбудив исключение *ош_связь* в точке вызова. Менее ясно, что должно произойти, когда прекращается вызывающий процесс. Если вызываемый процесс рандеву еще не принял, то проще всего отменить вызов, оставляя при этом вызываемый процесс в неизвестности относительно возникновения исключительной ситуации. Если же, однако, в момент прекращения вызывающего процесса рандеву находится в состоянии развития, то для согласования с предыдущим случаем кажется разумным возбудить в вызываемом процессе исключение о*ш_связь.* Это решение нехорошо тем, что затем подпрограмма исключений в операторе **accept** может попытаться продолжить связь с прекращенной вызывающей задачей. Еще хуже, если оператор **accept** вложен в другой оператор **accept**, так как тогда вызываемый процесс вполне мог получить одновременно два исключения о*ш_связь.* Этих проблем можно не решать, если принять специальное правило, что исключение о*ш_связь* возбуждается не в операторе **accept**, а вне его в объемлющем блоке. Это решение, однако, не очень аккуратно, и может быть лучше вообще не допускать такой ситуации, запретив прекращать вызывающий процесс во время рандеву. Вместо этого можно активацию исключения *отключить* и вызов процедуры *прекратить* отложить до тех пор, пока не завершится рандеву. Это решение привлекательно тем, что оно упрощает разработку процесса-служащего и разработку всей системы в целом. Кроме того, с точки зрения общих принципов, можно доказывать, что не нужно требовать от процесса-служащего, чтобы он разбирался в ситуации неожиданного завершения своих клиентов.

Наше обсуждение исключений в параллельных процессах было по необходимости кратким. Это очень сложный вопрос, до сих пор являющийся предметом исследований. Основные выводы состоят в том, что исключения между параллельными процессами могут вызывать серьезные последствия из-за их асинхронной и непредсказуемой природы. Поэтому их использование должно быть ограниченным, а их семантика — по возможности наиболее простой; в противном случае программы могут стать неприемлемо трудными для понимания и доказательства.

Программируя на практике, всегда лучше завершать процессы с помощью посылки сообщений по стандартным каналам связи с указанием им завершить самих себя. Использование исключения *отключить* и процедуры *прекратить* нужно зарезервировать для специальных случаев «неконтактных разбойничьих» процессов, которые должны быть завершены принудительно.

8.3.6. Аспекты реализации

Основное требование к механизму обработки ошибок в языке реального времени состоит в том, чтобы его можно было реализовать так, чтобы уменьшение скорости работы программ

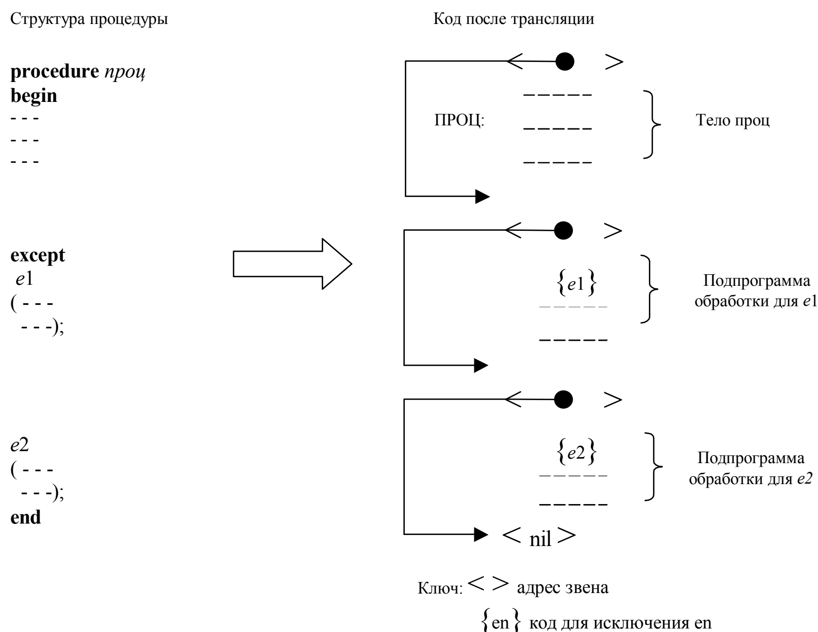


Рис. 8.2. Реализация подпрограмм обработки исключений.

происходило только в случае фактического использования этого механизма. В данном разделе для демонстрации осуществимости ограничения нулевых издержек времени работы будет приведена общая схема возможной стратегии реализации описанного выше механизма обработки исключений.

Пусть приведенная на рис. 8.2 схема кода программы получена при трансляции процедуры *проц* с локальными подпрограммами обработки исключений. Основным в этой схеме является то, что, перед тем как запомнить входную точку процедуры, запоминается указатель, содержащий адрес первой подпрограммы исключений. Вторая подпрограмма связывается в цепочку с. первой запоминанием указателя непосредственно перед первой подпрограммой. Точно таким образом связываются в цепочку все подпрограммы, поэтому, зная адрес входа в процедуру, можно найти все подпрограммы исключений, описанные в этой процедуре. В заголовке каждой подпрограммы

находится уникальный код (или коды), идентифицирующий исключение (или исключения), которое обслуживает данная подпрограмма.

Если данная схема используется для всех процедур, обработка всех исключений может быть полностью выполнена с помощью операторов исключения (**raise**). Для каждой процедуры вместе с записью ее активации будет храниться адрес возврата к вызывающей процедуре. Этот адрес возврата будет указывать на команду, находящуюся после команды вызова процедуры, что позволяет найти саму команду вызова. Эта команда вызова будет содержать адрес входа в вызываемую процедуру, что позволит найти подпрограммы обработки для этой процедуры. Таким образом, действия оператора исключения можно приблизительно описать следующим образом:

(1) Выбрать адрес возврата вызывающей процедуры...
(2)... отсюда установить адрес входа текущей процедуры.
(3) Просмотреть цепочку подпрограмм обработки и выбрать ту, которая соответствует возбужденному исключению.
(4) Если такой подпрограммы не найдено, выполнить возврат к вызывающей процедуре (т. е. удалить из стека запись текущей активации) и все повторить с шага (1).
(5) Когда подпрограмма наконец найдена, выполнить ее.

Такая стратегия реализации может оказаться очень медленной, но это не важно. Существенным здесь является то, что она не дает издержек времени работы и, таким образом, демонстрирует осуществимость принципа реализации подпрограмм исключений с данным ограничением.

8.4. ОБРАБОТКА ОШИБОК В ПРАКТИЧЕСКИХ ЯЗЫКАХ

В прежних языках реального времени оператор **goto** является основным механизмом обработки ошибок. Параметризация меток, на которые осуществлялся уход, обеспечивалась двумя различными способами. Фиксированные метки можно сгруппировать в массивы с помощью описания *переключателя,* причем значение конкретной метки вычисляется во время работы программы при использовании индекса в операторе **goto,** т. е.

goto $s[i]$

выполнит переход на i-ю метку в массиве переключателя s. Этот механизм используется в языке JOVIAL (Шоу, 1963) и языке CORAL (Вудвард и др., 1970). Меточные переменные, с другой стороны, можно реализовать так, как описано в разд. 8.2, что дает несколько более гибкий механизм, хотя в исключительных случаях он менее надежен. Этот второй подход принят в языке RTL/2 (Барнес, 1976, и гл. 11).

Интересно отметить, что в Модуле (Вирт, 1977а, и гл. 12) предусмотрена невозможность использования какой бы то ни было формы оператора **goto;** вместо этого программист должен полагаться на процедурные параметры и обычные операторы управления при обработке ошибок. Разработчики Модулы называют это особой формой эксперимента программирования без **goto.** В конечном счете эксперимент показывает, что там, где нужно выполнять большую работу по проверке ошибок и их обработке, язык оказывается недостаточно гибким. Очень часто текст программы претерпевает большие искажения в связи с дополнительным кодом, который нужен для обработки ненормальных условий.

Механизмы обработки исключений являются относительно новым средством в проектах языков программирования. Наиболее значительными примерами существующих языков, предоставляющих исключения, являются PL/1 (IBM, 1971) и Mesa (Гешке и Сатертвайте, 1977). Оба этих языка обеспечивают обработку исключений в достаточно общей форме, что позволяет использовать их в качестве стандартного программистского средства, а не только исключительно для обработки ошибок. Однако опыт работы с этими языками показывает, что расширенное использование исключений приводит к получению неясных программ, а сложность самого механизма заставляет многих программистов относиться к нему с недоверием.

Более ограниченная форма обработки исключений, описанная в разд. 8.3, базируется на предложениях Брона и Фоккинги (1977). В этом подходе все исключения рассматриваются как условия завершения и рассчитаны на использование только для обработки ошибок. От «особо надежных» языков требовалось, чтобы они обеспечили обработку исключений именно такого типа и все результирующие проекты оказались достаточно похожими. Механизмы, предоставляемые языком Green (Ихбиа, 1979) и результирующим языком Ада (см. гл. 13), являются по существу механизмом, который описан в этой главе. Язык Yellow (SRI, 1978) предоставлял даже еще более ограниченную систему: локальные параметры прекращенной процедуры были недоступны в локальной подпрограмме обработки исключений. Кроме этого, язык Yellow предоставлял различные механизмы для исключений, возбужденных программно, и для исключений, возбужденных в процессе выполнения; последний имел форму подпрограммы обработки аварийных ситуаций.

В конце следует подчеркнуть, что недостаток опыта использования подпрограмм обработки исключений в практической области заставляет подходить к ним особенно осторожно при разработке языка. Вполне может оказаться, что опыт работы

с языком Ада явится толчком для разработки других подходов. В частности, развитие систем машинного оборудования в форме схем автоматического восстановления состояния вполне может привести к появлению более мощных механизмов, которые одновременно окажутся и более удобными для использования.

Глава 9. ПРОЕКТИРОВАНИЕ ЯЗЫКА

В первой главе, рассмотрев характеристики программного обеспечения реального времени с учетом более общих потребностей современной практики проектирования программного обеспечения, мы установили совокупность предъявляемых к языку требований. Затем каждое из этих требований было рассмотрено более подробно, при этом в качестве основы процесса проектирования языка использовались шесть основных критериев проектирования. Такими критериями являются надежность, удобочитаемость, гибкость, простота, мобильность и эффективность. В данной главе будет сделана попытка просуммировать все основные результаты проведенного обсуждения и рассмотреть их интеграцию в рамках одного языка программирования.

В первых пяти разделах этой главы делается общий обзор тем, которые были уже рассмотрены ранее; к их числу относятся типизация данных, структура программ, мультипрограммирование, программирование устройств низкого уровня и обработка ошибок. Хотя мы считаем, что эти вопросы являются вопросами первоочередной важности, существует тем не менее и ряд других важных моментов, которые не были затронуты. Они кратко рассматриваются в разд. 9.6 в рубриках: раздельная компиляция, инициализация переменных, ввод-вывод высокого уровня и порождающие программные единицы. В разд. 9.7 мы обращаемся к вопросу выбора подходящего множества свойств языка реального времени; речь идет о том, что нужно в язык вводить и что, возможно, не следует торопиться вводить. В заключение, в разд. 9.8 обсуждаются некоторые из общих правил, которые следует соблюдать, рассматривая проектирование языка как единое целое.

9.1. ТИПИЗАЦИЯ ДАННЫХ

Как было впервые отмечено в гл. 1 и затем неоднократно подчеркивалось в последующих главах, система типизации данных языка является краеугольным камнем его внутренней

надежности. Более того, гибкость и ортогональность возможностей спецификации данных в программе оказывает очень большое влияние как на выразительную мощь языка, так и на удобочитаемость написанного на этом языке текста. Поэтому совершенно ясно, что прежде всего нужно выбрать безупречную систему типизации языка.

Тип объекта данных специфицирует область значений, которые может принимать объект, и множество операций, которые можно применять к нему. Относящаяся к типу информация используется также компилятором для правильного представления объекта в памяти и для генерации соответствующей последовательности машинных команд для каждой из операций высокого уровня. Во второй главе было показано, что систему типизации можно характеризовать с помощью ее «силы». Слабая типизация (разд. 2.2.1) имеет дело главным образом с обеспечением того, чтобы все операции над объектами данных были согласованы на машинном уровне. Как правило, область допустимых значений для скалярных типов ограничивается лишь длиной слова базового машинного оборудования, а множествам операций для каждого скалярного типа разрешается пересекаться, что представляет большую гибкость программирования. Например, чтобы облегчить манипулирование с битами, к целым операндам разрешается применять логические операции. Характеризуясь очень большой гибкостью и простотой понимания, слабая типизация безусловно ненадежна. Сильная типизация (разд. 2.2.2) накладывает больше ограничений на использование объектов данных. Присваивать значения объектам можно только в случае эквивалентности их типов, а множества операций для различных типов логически не пересекаются. Поэтому сильная типизация является более надежной за счет некоторой потери гибкости, что должно компенсироваться дополнительными языковыми особенностями.

Здесь следует подчеркнуть, что такая оценка системы типизации в терминах ее силы является слишком упрощенной. Сила типизации в практических языках бывает самой различной: очень слабой, средней и очень сильной. Однако здесь нас не интересует вопрос точного определения силы типизации. Важным является то, что система сильной типизации обеспечивает надежные проекты языков.

Учитывая тот факт, что для языка реального времени надежность является самой важной характеристикой, при проектировании языка реального времени мы, безусловно, должны стремиться к предоставлению системы сильной типизации. В гл. 2 мы следовали этой стратегии, пытаясь разработать очень сильную подсистему типизации. В ходе этого было введено в рассмотрение два очень важных соображения.

Во-первых, тип должен специфицировать логически связанный класс объектов вне связи с остальными классами объектов, даже если у них имеется общий базовый тип. Например, в программе может содержаться целая переменная, представляющая индекс массива, и другая целая переменная, представляющая день месяца. Такие объекты должны иметь отличающиеся типы, чтобы можно было предотвратить выполнение таких незаконных операций, как присваивание индексного значения объекту, представляющему день месяца. Для обеспечения этого достаточно разрешить образовывать новые типы из существующих (разд. 2.2.3). Далее, должна иметься возможность с помощью метода определения эквивалентности типов различать производные типы с общим базовым типом. В связи с этим предпочтительнее выбрать систему, основывающуюся на именной эквивалентности, а не на структурной эквивалентности (разд. 2.2.4).

Во-вторых, надежность увеличивается еще больше, когда диапазон допустимых значений для каждого объекта данных ограничивается в точности требуемыми значениями. Например, если некоторая целая переменная используется для представления секунд, то диапазон ее значений можно было бы специфицировать как содержащий целые от 0 до 59. Такие ограничения можно вводить либо с помощью образования нового производного типа со специфицируемым диапазоном значений (разд. 2.2.6), либо описывая подтип (разд. 2.2.7). Второй способ позволяет накладывать ограничения диапазона на подмножество объектов типа, по-прежнему разрешая этим объектам свободно участвовать в операциях с другими объектами этого типа.

В результате введения в язык системы типизации на основе этих соображений надежность программы увеличивается, а дополнительная точность, с которой специфицируются данные, увеличивает легкость чтения программы, что помогает ее лучше понимать и поддерживать. Впрочем, имеются и некоторые недостатки, которые не следует оставлять без внимания.

Во-первых, система типизации сама по себе является достаточно сложным механизмом, так что она может оказаться трудной для понимания и еще более трудной для правильного использования. Во-вторых, очень сильная типизация оказывает влияние на все аспекты разработки программы. Особенно это сказывается, в частности, на интерфейсе процедур и модулей; поэтому программист должен быть особенно внимательным при типизации своих данных, если он хочет эффективно использовать общие ресурсы. Все это ведет к более высокой стоимости обучения программистов и необходимости поручать работу программистам с более высокой квалификацией. Для больших

проектов значительно более высокое качество конечного программного продукта вполне оправдывает эти расходы. Однако для маленьких проектов оправдать эти расходы может оказаться затруднительно и более подходящим здесь оказывается язык попроще с менее сильной типизацией (например, Паскаль).

Другим аспектом типизации является спецификация базовых скалярных типов, с помощью которых программист может строить свои собственные типы данных. Во-первых, было показано, что понятие перечислимого типа является мощным средством для описания нечисловых скалярных данных, включая определение предопределенных символьных и логических типов (разд. 2.3). Во-вторых, многие из проблем мобильности, связанные с числовыми типами, могут быть решены при помощи ранее определенного понятия ограничения диапазона и спецификации точности в случае вещественных типов (разд. 2.4).

Переходя от скалярных типов к составным типам данных, в гл. 3 мы рассмотрели спецификации структурных данных. Были описаны две основные формы структурирования: массивы (разд. 3.2) и записи (разд. 3.3). Кроме того, были описаны множественные типы для представления множественных значений (разд. 3.4) и обсуждены указатели, обеспечивающие построение динамических структур данных (разд. 3.5). И здесь для обеспечения полного набора описанных возможностей требуется система значительной сложности. Записи и массивы, безусловно, очень важны, и если предполагается реализация схемы с сильной типизацией, то трудности, связанные с обеспечением механизмов подтипизации для этих структур, на практике, по-видимому, неизбежны. Доказать необходимость множеств и указателей, впрочем, труднее. Множества можно моделировать с помощью логических массивов, а динамические структуры данных можно моделировать с помощью массивов записей. Хотя в результате может произойти некоторая потеря эффективности и гибкости получающееся в результате упрощение языка и его реализации можно считать адекватной компенсацией.

И наконец, следует повторить еще раз, что в этой книге рассматривались только статические схемы типизации. Статическая типизация обладает тем свойством, что типы всех объектов данных можно определить во время компиляции. Если будущие поколения разработчиков машинного оборудования введут в свои машины встроенные механизмы проверки типа, то системы динамической типизации в языках реального времени могут стать практически реальными. В случае когда такое произойдет, потребуется полный пересмотр обсуждаемой темы.

9.2. СТРУКТУРА ПРОГРАММ

Все классические возможности, предоставляемые языком для структурирования программ, можно разделить на два уровня. На первом уровне структуры управления нужны для спецификации последовательности основных программных действий. На втором уровне структуры нужны для группировки множеств логически соотносящихся действий в одну единицу. Рассматриваемые вместе эти возможности образуют базовые средства методологии, известной под названием «структурного программирования». Эта методология предполагает процесс разработки программ методом сверху вниз, когда на каждом уровне программа специфицируется в терминах множества абстрактных действий. Каждое абстрактное действие уточняется затем на следующем уровне его спецификации в терминах следующих (чуть менее) абстрактных действий. Для поддержания этой, ставшей в настоящее время классической методологии необходимы средства программирования, которые, таким образом, являются средствами абстракции управления.

Использование абстракции управления является мощным методом программирования, но тем не менее недостаточным для «обуздания» сложности очень больших систем. Вследствие этого появились другие методологии, базирующиеся на абстракции данных; в них разработка программ осуществляется в большей степени с помощью последовательного уточнения типов абстрактных данных, а не абстрактных действий. Для поддержания такой методологии язык должен обеспечивать конструкции для объединения данных и процедур в одну самостоятельную единицу. Получающаяся в результате конструкция должна обладать интерфейсом, который специфицирует, какие имена этой единицы видимы из внешнего мира. Как правило, эти имена будут именами процедур, обрабатывающих заключенные в единице данные; таким образом, поскольку не разрешены непосредственные обращения к данным, гарантируется их целостность относительно внешнего воздействия. Более того, фактическая реализация того, как происходит оперирование с данными, пользователю не видна, что и обеспечивает требуемую абстракцию.

Классические возможности структурирования программ были обсуждены в гл. 4. На нижнем уровне проект операторов управления достаточно бесспорен. Было предложено множество желательных характеристик и затем описаны примерные схемы для условного оператора, операторов выбора и повторения (разд. 4.2). Был установлен следующий основной принцип: все операторы управления должны иметь явное завершающее ключевое слово, чтобы иерархические структуры управления можно было представлять ясно и недвусмысленно. Несколько спор-

ным предложением было требование введения оператора выхода для обеспечения ненормального завершения циклов. Введение этой возможности оправдывается тем, что она часто приводит к более простым и более эффективным структурам. Кроме этого, она делает менее необходимым введение потенциально значительно более опасного оператора goto. Тем не менее она противоречит принципу структурного программирования один-вход/один-выход, и поэтому включение этой возможности в проект языка не представляется очевидным. Впрочем, принятые в этой области решения, как правило, не оказывают большого влияния на язык в целом. Если базовое множество операторов управления задано, то их точная форма является на самом деле не более чем вопросом «косметики». Основные вопросы проектирования, которые должны быть рассмотрены в связи с классическим структурированием программ, относятся к возможностям описания процедур и функций. В разд. 4.4 были освещены три главные проблемы. Вопервых, классы параметров можно было бы специфицировать абстрактно в терминах направления потока данных, т. е. в, из; или в/из; с другой стороны, их можно было бы специфицировать в терминах того, как они реализуются, т. е. копированием или по ссылке (разд. 4.4.2). У каждого из этих подходов имеются свои недостатки, и, по-видимому, ни один из них не имеет явных преимуществ перед другим. Во-вторых, как из требования ортогональности, так и из необходимости обеспечивать иерархическую разработку следует, что нужно допустить определения вложенных процедур. В этих случаях не ясно, какие применять правила именной области действия (разд. 4.4.3). Традиционно для процедур вводятся правила открытой области действия, т. е. все описанные во внешних блоках имена автоматически доступны во вложенной процедуре. Другой подход состоит в таком определении процедур, когда их области действия закрыты, т. е. когда в обычных условиях никакие внешние имена во вложенной процедуре не видимы. При необходимости доступа к внешнему объекту его имя должно быть явно описано оператором **use** в заголовке процедуры. Если применяется второй подход, то программы становятся яснее, увеличивается их надежность и упрощается оптимизация механизма вызова процедур. Однако и здесь увеличивается нагрузка на программиста. В-третьих, язык программирования должен быть расширяемым в том смысле, что типам данных, определенным пользователем, должны предоставляться такие же права, какие имеют и предопределенные типы. Из этого следует, в частности, что должна иметься возможность совмещения определения процедур; например, стандартная функция *писать* (x), где x - объект любого предопределенного скалярного

типа, может быть расширена так, чтобы включать определенные пользователем типы (разд. 4.5.2). Продолжая это рассуждение дальше, заметим, что также необходима возможность совмещать и операции. Основная проблема в совмещении, не считая некоторых технических трудностей, заключается в том, что совмещение дополнительно усложняет язык и затрудняет его реализацию.

Механизмы абстракции обсуждались в гл. 5 в форме специальных конструкций, называемых модулями. Модуль образует вокруг содержащихся в нем объектов нечто вроде забора, вне модуля видимы только те объекты, которые описаны в спецификации интерфейса. Про такие объекты говорят, что они экспортируются. Сам модуль обрабатывается только во время компиляции и, таким образом, время работы готовой программы не увеличивается. Принимая во внимание его эффективность при модуляризации больших программ и оказываемую им поддержку при использовании абстракции данных, его включение в проект современного языка реального времени фактически следует считать обязательным.

При разработке модулей все же возникают определенные трудности, большинство из них связано с экспортом перечислимых типов и структурных типов. По существу, нужно задать достаточно много информации пользователю экспортируемого типа, чтобы он имел возможность описывать и использовать объекты этого типа со скрытой внутренней структурой. Решения этих проблем приведены ранее (разд. 5.3.2), но следует заметить, что снова возникают трудности, если мы хотим разрешить приватную часть модуля компилировать отдельно от соответствующих спецификаций интерфейса (см. разд. 9.6.1).

9.3. МУЛЬТИПРОГРАММИРОВАНИЕ

Для того чтобы описать и построить систему программного обеспечения в терминах набора совместно исполняемых задач, требуются возможности мультипрограммирования. Эти возможности должны обеспечить спецификацию программных процессов и предоставить им некоторые средства общения и синхронизации друг с другом.

В гл. 6 было описано несколько подходов к мультипрограммированию. Традиционный подход состоит в представлении каждого процесса отдельной программой (разд. 6.2). Общение обеспечивается тем, что каждой программе разрешается доступ к области общей памяти; использование семафоров при этом обеспечивает взаимное исключение. Синхронизация между программами достигается с помощью механизма сигнализации.

Этот подход особенно привлекателен тем, что в самом языке не обязательно должна иметься какая-либо специфическая

мультипрограммная конструкция. Все требуемые возможности предоставляются многозадачной операционной системой, которая осуществляет интерфейс с каждой программой задачи посредством множества обращений к стандартным процедурам, таким, как *оградить* и *освободить* для семафоров и *ждать* и *послать* для сигналов. Хотя этот подход и привлекателен из-за своей простоты, он имеет ряд недостатков, которые делают его неприемлемым в проекте современного языка реального времени. Во-первых, связь и синхронизация с помощью семафоров и сигналов являются средствами очень низкого уровня, не являющимися достаточно надежными. Семафоры, в частности, часто могут использоваться не так как нужно, а последствия такого неправильного использования могут с трудом поддаваться исправлению. Кроме этого, их применение может сильно затруднить понимание работы программ. Во-вторых, так как каждый процесс компилируется отдельно как индивидуальная программа, компилятору ничего не известно о структуре межпроцессного взаимодействия, и поэтому проверки во время компиляции невозможны. Вследствие этого все проверки должны выполняться операционной системой во время работы программы, что приводит к неизбежному замедлению ее работы. В-третьих, построенные таким способом системы редко бывают мобильными, так как они зависят от характеристик используемой операционной системы, которые зачастую тоже машинно-зависимы.

Современный подход к мультипрограммированию состоит в введении специальных конструкций для спецификации процессов и межпроцессной связи и синхронизации в самом языке. Это ведет к значительно более высокой надежности, так как компилятор может проверить, что спецификации интерфейса процесса строго согласованы. Кроме того, так как язык позволяет описывать не только индивидуальные процессы, но и всю мультипроцессную систему, общая структура системы становится более ясной и более легкой для понимания.

Процесс может быть описан с помощью нотации, аналогичной нотации, которая используется для описания процедур (разд. 6.3). Основная возникающая здесь проблема относится к тому случаю, когда возбуждается несколько копий одного и того же процесса; чтобы управлять процессом, должна иметься возможность ссылаться на каждое индивидуальное возбуждение. Это особенно важно в непрерывно работающих системах, где главный процесс должен иметь возможность прекращать «разбойничьи» процессы. Два главных механизма идентификации процессов основываются соответственно на индексах процессов и указателях на процессы. Эти механизмы аналогичны механизмам доступа к данным с помощью индексов и указате-

лей, при оценке их относительных достоинств можно применять те же самые критерии. Использование индексов требует, чтобы общее число активаций типа процесс было ограниченным, поэтому в этом случае требования к памяти известны во время компиляции. При использовании указателей память при каждой активации процесса определяется динамически, поэтому требуется более сложная схема управления памятью, но в результате система получается более гибкой.

Надежность мультипроцессной системы зависит главным образом от того, как процессы осуществляют связь друг с другом. Два подхода высокого уровня к межпроцессной связи были описаны в гл. 6. Мониторы позволяют процессам общаться друг с другом с помощью областей общих данных. Монитор является разновидностью модуля, он также объединяет в одно целое общие данные и множество процедур интерфейса, с помощью которых осуществляется доступ к данным. Монитор обеспечивает выполнение интерфейсных процедур в режиме взаимного исключения, гарантируя целостность общих данных. Так как он обладает теми же самыми свойствами абстракции данных, что и обычный модуль, с его помощью можно построить хорошо структурированные и надежные мультипроцессные системы. Впрочем, и у него имеются недостатки; таких недостатков четыре. Во-первых, синхронизация между процессами по-прежнему основывается на примитивном понятии сигнала. Хотя присущий использованию сигналов низкого уровня риск нарушить правильную работу системы уменьшается в связи с ограничением области их использования только внутри монитора, само привлечение сигналов представляется нарушением баланса между механизмом общения высокого уровня и механизмом синхронизации низкого уровня. Во-вторых, когда внутри монитора находится несколько процессов, не имеется простого способа, с помощью которого процесс синхронизации мог бы заново запустить их всех одновременно. В-третьих, вложенные обращения к монитору представляют трудности с точки зрения как семантики языка, так и его реализации. В-четвертых, так как монитор рассчитан на обеспечение связи через области общих данных, с его помощью нельзя получить естественное описание для систем, реализованных с использованием архитектуры распределенных вычислительных машин.

Второй подход к межпроцессной связи, являющийся подходом высокого уровня, базируется на понятии рандеву (разд. 6.5). Рандеву объединяет в один механизм взаимное исключение и синхронизацию. При использовании рандеву процессы общаются друг с другом, сначала синхронизуясь, а затем обмениваясь данными непосредственно, без явного использования областей общих данных. Таким образом, рандеву в

отношении межпроцессной связи и синхронизации является подходом очень высокого уровня, который не имеет ни одного из недостатков монитора. Его единственная отрицательная сторона состоит в том, что его значительно труднее реализовывать, а используемые решения требуют, как правило, больше процессов в связи с необходимостью раздваивать асинхронные процессы с помощью активных буферных процессов, а не пассивных мониторов. Впрочем, последняя проблема может быть в какой-то степени решена с помощью оптимизирующего компилятора, так как очень часто буферные процессы можно преобразовать в пассивные структуры мониторного типа. Еще одна небольшая трудность, касающаяся механизма рандеву, состоит в том, что он менее эффективен при программировании устройств низкого уровня (см. следующий раздел). Тем не менее ни один из этих недостатков не является серьезным, и механизм рандеву, безусловно, следует считать главным достижением в области проектирования высоконадежного языка мультипрограммирования.

9.4. ПРОГРАММИРОВАНИЕ УСТРОЙСТВ НИЗКОГО УРОВНЯ

Так как главная характеристика вычислительных систем реального времени состоит в том, что они часто содержат несколько устройств ввода-вывода специального назначения, кажется странным, что очень мало проектов языков реального времени содержат возможности программирования низкого уровня. В гл. 7 были рассмотрены требования к программированию устройств низкого уровня непосредственно в языке высокого уровня. Было показано, что требуются три вида возможностей.

Во-первых, должна иметься возможность доступа к индивидуальным битам и битовым полям регистров устройств ввода-вывода, расположенным как в памяти, так и в специальных массивах ввода-вывода (разд. 72). Для этого достаточно простого разрешения специфицировать при описании адрес, в котором будет храниться переменная. Последующие присваивания такой переменной затем интерпретируются компилятором как операции писать в соответствующий регистр, а использование значения переменной в выражении — как операция читать из регистра. Доступ к индивидуальным битам и битовым полям можно обеспечить, разрешив применять логические операции и операции сдвига к скалярной переменной, связанной с этим регистром, или более структурным способом, используя комбинированные, регулярные и множественные типы вместе со связанной спецификацией отображения или представления. Второй подход лучше, так как он способствует получению более ясных программ и дает возможность воспользоваться всеми преимуществами сильной типизации.

Во-вторых, язык должен содержать некоторые средства для представления в программной структуре асинхронного поведения операций устройств ввода-вывода. Лучше всего это обеспечить с помощью мультипрограммной схемы, в которой программы управления устройствами ввода-вывода трактуются как процессы программного обеспечения (разд. 7.3). Интерфейс между ними реализуется с помощью обычного механизма межпроцессной связи. В гл. 7 использование монитора в этой роли было показано на нескольких примерах (разд. 7.4). По-видимому, монитор является особенно удобным средством, так как механизм сигнализации низкого уровня, применяемый для синхронизации процессов, очень естественно соответствует механизму сигнализации машинного оборудования, используемому устройствами ввода-вывода, т. е. прерываниям. Таким образом, в мониторе ожидание сигнала от машинного процесса ввода-вывода непосредственно переводится в ожидание прерывания. Аналогичный эффект может быть получен при использовании механизма рандеву, где прерывание связывается с обращением машинного оборудования к оператору приема; но это несколько менее естественно, здесь для получения сравнимой эффективности требуется значительная оптимизация.

В-третьих, любая программа, которая обращается к устройствам машинного оборудования непосредственно, неизбежно становится машинно-зависимой. Впрочем, некоторая степень мобильности может быть сохранена, если строго следить за тем, чтобы все такие машинно-зависимые части заключались в модули ввода-вывода (разд. 7.2.3). Это локализует конструкции программирования низкого уровня в небольших, легко идентифицируемых областях программного текста и упрощает перекодировку в тех случаях, когда становится необходим переход на другую машину.

Включение этих возможностей программирования устройств низкого уровня в параллельный язык высокого уровня увеличивает сложность языка лишь в очень небольшой степени, так как в основном требуются лишь небольшие расширения существующих конструкций. Зато в качестве компенсации появляется возможность построения системы программного обеспечения реального времени непосредственно на «голой» машине без обращения к вставкам машинного кода или библиотечным программам, написанным в машинном коде. Поэтому включение рассмотренных возможностей в проект языка реального времени кажется вполне целесообразным.

9.5. ОБРАБОТКА ОШИБОК

Из требований высокой надежности и непрерывного функционирования систем реального времени следует, что язык ре-

ального времени должен обеспечивать механизмы реакции на условия ошибок и программирования действий по восстановлению после ошибок. В гл. 8 были обсуждены базовые понятия обработки ошибок и перечислены пять основных требований к механизму обработки ошибок. Этими требованиями являются простота, ясность, временные издержки только при фактическом использовании, одинаковая обработка программно-обнаруживаемых ошибок и машинно-обнаружизаемых ошибок и возможность программирования действий восстановления.

Традиционные механизмы обработки ошибок включают использование параметров ошибок в обращениях к процедурам и (или) в глобальных операторах **goto** (разд. 8.2). Оператор **goto** является мощным механизмом и вместе с меточными переменными удовлетворяет операционным требованиям к механизму обработки ошибок. Главный недостаток оператора **goto** состоит в том, что он очень низкого уровня и ненадежен, а использующие его программы, которые выполняют большую работу по восстановлению, как правило, бывают крайне непонятными.

Основное содержание гл. 8 связано с разработкой более структурного подхода обработки ошибок, базирующегося на идее исключений, и подпрограмм их обработки (разд. 8.3). Исключение представляет разновидность флажка ошибки, который может находиться либо в возбужденном, либо в невозбужденном состоянии. В возбужденном состоянии он указывает на появление условия ошибки. В этом случае выполнение текущей подпрограммной единицы заканчивается и, если в этой единице имеется подпрограмма обработки данного исключения, управление автоматически передается этой подпрограмме. Таким образом, подпрограмма обработки является заменяющей последовательностью кодов, которая выполняется вместо оставшегося кода подпрограммы. Подпрограмма обработки имеет возможность доступа к любому объекту из незавершенной подпрограммы и, следовательно, может быть использована для программирования действий восстановления после ошибки. Если в текущей подпрограммной единице такой подпрограммы обработки найти нельзя, то данная единица завершается и управление передается вызывающей подпрограмме, в которой в точке обращения перевозбуждается то же самое исключение. Этот процесс продолжается до тех пор, пока не будет найдена подпрограмма обработки. Отсюда следует, что данный механизм связывает исключение с соответствующей подпрограммой его обработки динамически и реализует абстрактное понятие области ошибки, описанное в разд. 8.3.1.

Разработанный в гл. 8 механизм обработки исключений удовлетворяет всем перечисленным выше требованиям к обработке

ошибок и обеспечивает возможность структурного подхода к программированию действий восстановления. Следует заметить, что можно ввести более сложные и мощные средства обработки исключений, если разрешать возвращать управление той подпрограмме, в которой было возбуждено исключение, после завершения подпрограммы обработки исключений. Однако такой механизм трудно реализовать аккуратно, и, что еще хуже, у программиста возникает соблазн использовать его в качестве общего программистского средства. Насильственное завершение подпрограммной единицы в случае возбуждения в ней исключения заставляет рассматривать исключения как условия ненормального завершения и таким образом ограничивает их использование только обработкой ошибок.

Несмотря на несомненные достоинства механизмов обработки исключений, им присущи некоторые недостатки, которые делают вопрос об их включении в проект языка довольно спорным. Во-первых, в то время как их операционную семантику для простых последовательных программ определить просто, ее определение становится менее ясным, когда исключения возникают между взаимодействующими параллельными процессами. Во-вторых, общая сложность подпрограмм обработки исключений значительно увеличивает сложность как языка, так и компилятора. В тех случаях, когда использование простого оператора **goto** достаточно применительно к данной прикладной области, может оказаться не просто оправдать это увеличение сложности из-за подпрограмм обработки исключений.

9.6. ДРУГИЕ АСПЕКТЫ ПРОЕКТИРОВАНИЯ ЯЗЫКА

В предыдущих разделах в общих чертах рассмотрены основные вопросы проектирования языка, которые были разобраны в данной книге. В этом разделе будут кратко обсуждены еще четыре дополнительных аспекта проектирования.

9.6.1. Раздельная компиляция

Возможность раздельной компиляции индивидуальных компонентов системы программного обеспечения в практическом языке реального времени необходима. Раздельная компиляция позволяет компилировать очень большие программы на машинах с достаточно скромными ресурсами памяти, не требует перекомпиляции всей программы при внесении одной модификации и упрощает разработку проекта, позволяя писать и отлаживать компоненты программного обеспечения независимо друг от друга. Любая программная единица, для которой можно определить интерфейс, может рассматриваться как объект раз-

дельной компиляции. В связи с этим в традиционных языках в качестве единиц компиляции обычно рассматриваются процедуры и функции с возможным привлечением также некоторого вида пула общих данных. В современных языках мультипрограммирования единицами компиляции могут быть также модули и процессы.

Можно было бы считать, что раздельная компиляция вовсе не является аспектом проектирования языка, а относится скорее к области реализации. Для случая процедур, функций и процессов это, вообще говоря, верно. Однако раздельная компиляция модулей выявляет трудности, которые влияют и на проект самого языка; учитывая то, что модули являются потенциально наиболее полезными для раздельной компиляции из всех программных единиц, возникающие при этом проблемы следует обсудить несколько подробнее.

Напомним, что модуль состоит из двух частей: спецификации интерфейса и приватного тела, которое реализует модуль. В идеальном случае пользователю модуля требуется включить в свою программу только спецификационную часть, оставив главное тело модуля для раздельной компиляции. К сожалению, здесь компилятору приходится сталкиваться с рядом трудностей. Например, предположим, что модуль экспортирует абстрактный тип данных *T,* тогда спецификация интерфейса задаст только имя *T,* разрешая пользователю описывать объекты типа *T* и работать с этими объектами с помощью операций, задаваемых модулем. Хотя пользователю требуется только имя типа, компилятору требуется больше сведений. В частности, ему должно быть известно, как велики объекты типа *T,* чтобы выделить для них память. Он может вычислить этот размер только тогда, когда ему известно, какова внутренняя структура типа; но эта информация скрыта в приватной части модуля.

Эта проблема имеет два решения. Во-первых, можно ограничить произвол в порядке компилирования программных единиц и потребовать, чтобы тела модулей компилировались прежде, чем они могут быть использованы. Затем компилятору можно предоставить доступ к базе данных, которая содержит полную информацию о необходимых компилятору экспортируемых типах. Это решение накладывает ряд ограничений на способ построения системы программного обеспечения и противоречит духу абстракции данных в том смысле, что вынуждает прибегать к проектированию методом снизу вверх, а не сверху вниз. Второе решение состоит во включении дополнительной информации в спецификацию интерфейса, что позволяет компилятору работать без обращения к реализационной части модуля. Другими словами, внутренняя структура экспортируемых

типов должна специфицироваться, даже если не предполагается, что пользователю модуля нужно что-либо знать об этой структуре. Это решение не очень красиво и ограничивает свободу реализации модуля в выборе произвольной внутренней структуры типа, которая не влияла бы на программы пользователя. Поэтому ни одно из этих решений не идеально, и проектировщики языка должны пойти на компромисс между абстрактными требованиями к модулю как к языковой конструкции и практическими требованиями к модулю как к единице компиляции.

9.6.2. Инициализация переменных

Наряду со спецификацией типа переменной в ее описании полезно также иметь возможность специфицировать ее начальное значение. Конечно, эта возможность не является совершенно обязательной, так как переменным всегда можно присвоить начальное значение явно операторами присваивания в начале соответствующего тела операторов. Тем не менее возможность явной инициализации в описании увеличивает ясность программы и уменьшает количество мест в программе, где обычно часто совершаются ошибки. Кроме этого, в случае глобальных переменных увеличивается эффективность, так как инициализацию можно выполнить во время загрузки.

С точки зрения проектирования языка главная проблема инициализации состоит в определении формы выражений, которые нужно разрешить для спецификации начального значения. Выбор по существу лежит между разрешением любого произвольного выражения, которое может содержать переменные и обращения к функциям, и разрешением только статических выражений, которые можно полностью вычислить во время компиляции. Этот выбор существенно влияет на природу языка в целом. Из разрешения произвольных выражений следует, что обработка описания составляет активную часть вычислительного процесса, а не сводится к пассивной спецификации объектов данных. Особенно неприятным следствием этого является то, что при обработке описаний могут возникнуть ошибки. При этом усложняется семантика обработки ошибок за счет таких факторов, как необходимость знать, какие объекты были обработаны успешно, а какие нет. При разрешении только статических выражений этих проблем не возникает, а гибкость ухудшается незначительно. В тех редких случаях, когда начальное значение переменной должно быть вычислено динамически, всегда можно использовать явный оператор присваивания.

9.6.3. Ввод-вывод высокого уровня

Спецификация стандартных возможностей для программирования операций ввода-вывода для стандартных периферийных устройств, таких, как консоли, диски и печатающие устройства, особенно трудна в области реального времени. Во многих существующих языках этот вопрос либо не учтен вовсе, что заставляет реализующих язык программистов придумывать свой собственный набор возможностей, либо решается с помощью встроенных возможностей, которые на самом языке выразить нельзя (ср. оператор *write* языка Паскаль).

Мы не имеем возможности подробно обсудить эту тему, но кратко остановиться на приемлемых стратегиях проектирования необходимо. Одной из характеристик приложений реального времени является то, что их уровень может быть самым различным. Так, например, вложенный контроллер может не требовать никакого ввода-вывода высокого уровня, тогда как система управления процессом с работающей в режиме on-line информационной базой данных может потребовать развитые возможности работы с файлами. Удовлетворение всех этих различных нужд при помощи включения встроенных возможностей представляется в значительной степени неразрешимой задачей. Если же оставить ввод-вывод высокого уровня полностью неопределенным, то аналогичные затруднения может вызвать проблема мобильности.

Одним из решений данного аспекта рассматриваемого вопроса является предоставление в проекте языка возможности программировать на самом языке гибкие и адекватные операции ввода-вывода высокого уровня. В частности, включение механизма абстракции, такого, как модуль, позволяет представлять файлы и другие устройства ввода-вывода как типы абстрактных данных со связанным множеством соответствующих операций. Проект языка может затем специфицировать стандартное множество возможностей ввода-вывода высокого уровня не в терминах встроенных языковых конструкций, а в терминах предопределенных модулей. Конкретные пользователи языка могут позднее расширить или сократить эти возможности, приспосабливая их к своим требованиям. При этом подходе определения ввода-вывода высокого уровня получаются гибкими и мобильными; поэтому его следует считать, по-видимому, наиболее соответствующим областям применения реального времени.

9.6.4. Порождающие программные единицы

Во всех предыдущих главах основное внимание всегда уделялось статическим, а не динамическим свойствам языка.

Вследствие этого возможности параметризации программных единиц ограничиваются входными значениями процедур и функций. Например, мы не можем написать стековый модуль, такой, как в разд. 53, где тип объекта можно параметризовать.

Одно из решений этой проблемы состоит в использовании хорошо знакомой идеи макрогенератора (см, например, Коул, 1976) для спецификации порождающих программных единиц. Порождающая программная единица специфицируется точно таким же способом, как и обычная единица, с тем исключением, что ей предшествует список имен, которые обозначают порождающие параметры. Порождающая единица не может быть выполнена обычным способом, она служит образцом для создания параметризованных копий единицы, где формальные порождающие параметры заменяются на фактические параметры. Этот процесс порождения копий соответствует макрорасширению и целиком происходит во время компиляции. Поэтому издержки времени работы программы не связаны с этим механизмом.

Хотя порождающие программные единицы можно специфицировать и реализовать как макроопределения, имеются серьезные соображения, почему такой подход следует считать неприемлемым. Основное использование порождения состоит в обеспечении гибких библиотечных возможностей. Поэтому маловероятно, чтобы обычному прикладному программисту могло потребоваться знать, как следует определять порождающую программную единицу; он должен знать только, как создать ее копию. В связи с этим компилятор должен обладать способностью проверять раздельно структуру порождающей единицы и структуру создания порожденной копии. В частности, нужно уметь гарантировать, что при создании пользователем копии порождающей единицы компилятор не выдаст сообщение о содержащейся в теле этой единицы ошибке. Для обеспечения этого проект возможностей порождающего определения должен быть таков, чтобы форма порождающих параметров ограничивала предоставление типов объектов и чтобы это указывалось в спецификации порождающей единицы. Должна иметься возможность проверить в дальнейшем, что текст порождающей единицы корректен в отношении ее спецификации параметров, так чтобы при соответствии заданных фактических параметров этим спецификациям процесс порождения копии дал в результате семантически правильную программную единицу. Другим преимуществом ограничения порождающих образцов в смысле представления только программных единиц, а не произвольных текстовых строк является то, что компилятор не обязательно должен реализовывать генерацию копий с по-

мощью прямолинейного макрорасширения. В некоторых ситуациях он мог бы выделить общие части различных копий, экономя рабочую память.

9.7. ВЫБОР СВОЙСТВ ЯЗЫКА

В предыдущих главах каждая из рассмотренных языковых областей обсуждалась отдельно, без учета глобальной схемы языка. Как правило, конструкции разрабатывались, начиная с относительно простых понятий в направлении более развитых. В каждом случае ставилась цель дать общее представление о состоянии дел в интересующей нас области проектирования языка. В данном и следующем разделах этой главы будет кратко обсуждена интеграция этих конструкций в рамках одного языка, что можно рассматривать как введение во вторую часть этой книги; наше обсуждение прослеживает развитие проектов реальных языков.

Ясно, что язык программирования, снабженный всеми рассмотренными современными механизмами, т. е. очень сильной типизацией, базирующемся на рандеву мультипрограммированием, модулями, совмещенными процедурами, подпрограммами обработки исключений и т. д., будет не только очень мощным, но также очень большим и сложным. Поэтому при обсуждении были сделаны некоторые попытки как-то обосновать включение каждого из этих механизмов в язык реального времени. По существу, в каждом случае обоснование упиралось в необходимость справляться с все возрастающими размером и сложностью современных систем реального времени. Впрочем, некоторые утверждают, что ценность языка следует оценивать скорее по тому, что было опущено, чем по тому, что было включено. Другими словами, небольшие компактные языки гораздо лучше больших разнообразных языков. Действительно, это утверждение не лишено смысла; стоит сравнить популярность Паскаля с его (академическим) соперником Алголом 68. Преимущества того, что язык маленький, безусловно, не следует недооценивать. Если для языка требуется большой компилятор, то развитие программного обеспечения на целевую машину может оказаться невозможным. Вместо этого приходится использовать перекрестную компиляцию с большой главной системой в качестве обслуживающей вычислительной машины. Такой вид разработки усложняет тестирование и отладку, и, что более серьезно, он делает крайне трудной последующее поддержание и сопровождение в связи с невозможностью перекомпиляции на целевой машине. Аналогично, если язык требует большую систему для прогона программы, то он будет менее привлекателен для небольших целевых машин, где

память может оказаться ограниченной. Кроме этого, небольшой язык легче изучить, чем большой.

Тем не менее, несмотря на безусловную привлекательность небольшого языка, в этой книге защищается та точка зрения, что современный язык реального времени должен включать возможности, позволяющие удовлетворить все требования к языку, которые были рассмотрены в предыдущих главах. Доводы в пользу небольшого языка основываются на чувствительности к ограничениям технологии машинного оборудования и искусства программистов. Эти ограничения были достаточно существенными десять лет назад, но в настоящее время они сомнительны и лет через пять, безусловно, перестанут существовать. Самым главным здесь является то, что маленький язык с ограниченным набором возможностей не может справиться со сложностью современных вычислительных систем реального времени. Даже развитие микропроцессоров достигает такого уровня, когда обусловленная проектированием сложность машинного оборудования требует для поддержки язык, обладающий всеми развитыми механизмами, которые мы описали (см., например, Ратнер и Латтен, 1981).

Поэтому можно предположить, что язык, который будет отвечать нуждам программистов в реальном времени в ближайшие десять лет, должен включать следующие возможности:

(1) Систему очень сильной типизации с возможностями определения производных типов и подтипов. Предопределенные скалярные типы должны включать целые, фиксированные и плавающие, логические и символьные типы. Должна иметься возможность специфицировать диапазон всех дискретных типов и точность вещественных типов. Следует разрешить пользователю определять перечислимые типы.

(2) Такие структурные типы, как записи, массивы и множества, с возможностями подтипизации массивов и записей. Не должно быть ограничений на типы компонент, а массивы можно описывать любой размерности. Любой объект данных может быть распределен динамически, на него можно сослаться с помощью указательного типа. Должны иметься конструкторы для обозначения составных значений всех структурных типов.

(3) Структуры управления *if* (если), *case* (вариант), *while* (пока) и *for* (для). Переменная цикла в операторе *for* может пробегать значения любого дискретного типа или элементы множественного типа.

(4) Блочную структуру. Операторы можно группировать в блоки для введения новой именной области действия с локально определенными переменными. К блокам должны быть применимы классические правила открытой области действия.

(5) Процедуры, функции и операции подпрограммных единиц. Описания этих единиц могут помещаться на любой глубине вложенности. Каждая единица должна обладать замкнутой областью действия, но доступ к именам, описанным во внешней области, должен быть возможен либо с помощью спецификации имени в точке его описания как проникающего (pervasive), либо с помощью включения имени в список использования в точке использования. Не должно быть ограничений на типы формальных параметров или типы результатов, возвращаемых функциями и операциями. Любая подпрограммная единица может быть совмещенной.

(6) Модули для обеспечения объединения констант, типов, переменных и подпрограммных единиц. Модуль должен обеспечивать спецификации интерфейса для указания имен всех объектов, экспортируемых из модуля и импортируемых в модуль. Описания модулей могут быть вложенными в другие программные модули или подпрограммные единицы.

(7) Процессы для представления параллельно выполняемых деятельностей. На активацию каждого процесса можно индивидуально ссылаться с помощью либо индекса, либо указателя. Процессы могут вкладываться в другие процессы. Должен иметься механизм рандеву для обеспечения взаимной связи процессов с возможностью программирования задержек и завершения по исчерпанию времени. Любой процесс может быть явно прекращен другим процессом.

(8) Возможности для программирования устройств низкого уровня. Любая переменная может быть помещена по конкретному адресу памяти машины или устройства ввода-вывода. Имеется возможность специфицировать точное представление в памяти любого типа. Прерывания могут ставиться в соответствие обращениям к операторам приема механизма рандеву.

(9) Механизм обработки исключений для структурной обработки всех эксплуатационных программно-обнаруженных ошибок.

(10) Возможность порождающих определений для спецификации параметризованных образцов процедур, функций, операций, модулей и процессов.

(11) Возможность раздельной компиляции.

(12) Стандартное множество процедур ввода-вывода высокого уровня, определяемых в языке. Должны иметься возможности ввода-вывода в рамках последовательного и произвольного доступа как с форматной обработкой, так и без нее.

Представленный список возможностей охватывает область свойств, которые должен обеспечивать современный язык реального времени, если мы хотим, чтобы он работал в больших

системах в настоящее время и в будущем.

Как было отмечено ранее, обширность такого языка может привести в замешательство собирающихся использовать его в той прикладной области, где вычислительное машинное оборудование имеет ограниченные характеристики. В этом случае имеются два очевидных решения. Первое решение состоит в определении подмножества, удобного для использования в маленьких системах, второе — в том, чтобы дать специальный маленький язык для малых систем. У первого решения то преимущество, что по-прежнему поддерживается стандартная языковая нотация; во втором случае имеется возможность разработать более сбалансированное множество языковых свойств.

В любом из этих случаев приведенный выше список свойств может быть сокращен в следующих разделах:

(1) Система типизации может быть сделана менее строгой за счет использования некоторых форм структурной эквивалентности, что делает избыточными производные типы.

(2) Можно опустить указатели, динамические массивы и регулярные подтипы (вариантные записи слишком полезны, чтобы их опускать).

(3) Можно было бы опустить описание операций и связанный с ним механизм совмещения (аналогичный выигрыш можно получить при использовании процедур и функций).

(4) Можно упростить возможности для мультипрограммирования, запретив активацию вложенных процессов и использование мониторов для межпроцессной связи.

(5) Подпрограммы обработки исключений можно заменить простым механизмом глобального **goto.**

(6) Можно опустить порождаемые программные единицы (для достижения аналогичного эффекта можно использовать отдельный макропрепроцессор).

Данное обсуждение выбора языковых особенностей для языка реального времени специально не содержит никакой специфики и детализации. Во второй части этой книги описываются некоторые примеры проектов практических языков, которые показывают, как выбор языковых особенностей делается на практике.

9.8. ОБЩИЕ ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ

Чтобы завершить обсуждение проектирования языков программирования реального времени, следует сделать несколько замечаний относительно общей структуры языка. Независимо от размера и широты языка, имеется несколько основных

правил, которые следует соблюдать, намереваясь создать хороший проект.

Во-первых, язык должен обладать ясным и однозначным синтаксисом. При этом чем меньше синтаксических правил требуется для описания языка, тем лучше. Когда синтаксис прост, язык изучать легче и он лучше запоминается; обычно это указывает на то, что проектировщики аккуратно интегрировали все языковые свойства в рамках единой синтаксической схемы. В качестве примера единого синтаксиса рассмотрим использование обозначения *a..b* для описания дискретного диапазона значений от *a* до *b.* В полностью унифицированном синтаксисе это обозначение должно использоваться всегда, когда нужно специфицировать диапазон. В этом отношении язык Паскаль предоставляет пример неунифицируемости; диапазоны, специфицируемые в описании типа (например, ограничения подтипа, индексные диапазоны и т. д.), задаются с применением данного обозначения, но в операторе for используется иной синтаксис, а именно

for *i*:=*a* **to** *b* **do**...

Этот добавочный синтаксис совершенно не обязателен, так как прежнее обозначение диапазона вполне удовлетворительно (заметим, что в Паскале приращение разрешается только с шагом 1), т. е.

for *i* **in** *a. .b* **do**...

Еще одно преимущество более однообразной нотации состоит в следующем. Пусть имеется описание

type *индекс* = *a. .b*;
var *i* : *индекс*;

тогда цикл **for** с переменной *i* в качестве индекса можно записать более естественно:

for *i* **in** *индекс* **do**...

Таким образом, однообразие обозначений часто увеличивает гибкость нотации языка и уменьшает количество синтаксических правил.

Во-вторых, должно быть по возможности меньше ограничений на способы комбинирования языковых понятий. Например, если конструкция массива задается в виде

type *a* = **array**[*диапазон_индекса*] **of** *тип_компонент*;

то тип компонент не должен ограничиваться. Часто бывает очень соблазнительно запретить использование некоторых типов компонент, чтобы обеспечить генерацию эффективного

объектного кода и упростить компилятор. Однако лучше вообще отказаться от таких ограничений и предупреждать программиста об относительной стоимости различных конструкций. Ограничение общности языковых конструкций делает язык более трудным для изучения и менее естественным при использовании.

В-третьих, все свойства языка должны быть ортогональны, т. е. они не должны пересекаться. По существу, это означает, что должен быть только один способ построения нужной конструкции. Например, нежелательно включать в язык одновременно и вариантные записи, и типы объединения. Хотя в некоторых ситуациях объединения полезнее вариантных записей, а в других ситуациях — наоборот, предоставляемые ими возможности одинаковы. И здесь ортогональность ведет к более простому проекту языка, который легче понимать и использовать.

И последнее, предоставляемые языком возможности должны быть сбалансированы. Баланс — это мера, которую трудно оценить численно, но которая хорошо чувствуется интуитивно. Сбалансированный язык — это язык, в котором все предоставляемые возможности имеют примерно одинаковый уровень мощности и общности. Ярким примером несбалансированного языка является BCPL (Рихардс, 1969)., в котором наряду с очень мощными языковыми особенностями имеются очень слабые возможности структурирования данных.

Этим кратким обсуждением более общих аспектов проектирования языка заканчивается первая часть нашей книги. У читателя должно сложиться хорошее представление об имеющемся у проектировщика языка выборе различных возможностей в каждой из основных областей разработки. Во второй части этой книги достаточно подробно исследуются три проекта практических языков, которые показывают, как эти идеи проектирования могут быть реализованы на практике. Описываемыми языками являются языки RTL/2, Модула и Ада. Они были выбраны по двум причинам. Во-первых, они все являются очень хорошими проектами языков; во-вторых, каждый из них представляет соответственно один из основных классов проектирования языков реального времени. RTL/2 является представителем традиционного стиля. Это чисто последовательный алголоподобный язык. У него система слабой типизации. Модула представляет современный подход. Он базируется на Паскале и включает возможности мультипрограммирования. У него система сильной типизации, абстракция данных обеспечивается с помощью модулей. И тем не менее это очень маленький и компактный язык, что ведет к тенденции ограничить его использование областью небольших встроенных систем. Ада

представляет наиболее полно все последние достижения в области проектирования современных языков общего назначения для программирования в реальном времени. Язык Ада включает по существу все самые развитые возможности, описанные в предыдущих главах; в истории программирования систем реального времени он, возможно, явится поворотным пунктом.

Часть II. РАЗРАБОТКА

Глава 10. РАЗРАБОТКА ЯЗЫКОВ РЕАЛЬНОГО ВРЕМЕНИ

В этой части книги будет показано, как рассмотренные в первой части принципы проектирования использовались на практике; будут подробно описаны три языка: RTL/2, Модула и Ада. Каждый из этих языков является вехой в эволюции проектирования языков реального времени, и каждый реализует самые лучшие принципы проектирования своего времени.

На рис. 10.1 приведено дерево семейства языков, показывающее взаимосвязь между нашими тремя языками и некоторыми другими языками, которые оказали наибольшее влияние на их проектирование. Из этого рисунка можно увидеть, что история их развития охватывает период около двадцати лет. За это время имели место две основные тенденции.

Во-первых, в 1960 году было опубликовано сообщение о языке Алгол 60, явившееся событием чрезвычайной важности в области проектирования языков. Алгол 60 был первым языком, предоставившим для описания алгоритмов согласованное и единообразное множество конструкций. В частности, он впервые предоставил множество явных операторов управления для представления последовательного выполнения операторов, итерации и выбора. Это дало возможность писать программы с четко различимой передачей управления. Более того, стало возможным разрабатывать программы методом сверху вниз с помощью последовательного уточнения абстрактных действий, что можно считать моментом рождения «структурного программирования». Большие преимущества такого подхода быстро получили всеобщее признание, и вслед за Алголом 60 за небольшое время появилось много новых проектов языков, базирующихся на его принципах. Хотя сам Алгол 60 не получил особо большого практического применения, многие из его производных языков использовались довольно широко. В 1968 году был опубликован главный пересмотренный проект Алгола 60 в форме Алгола 68 (Ван Вейнгаарден и др., 1975). Язык Алгол 68 обладал основной схемой Алгола 60, его общность и мощь были увеличены за счет дальнейшего развития понятий

выражения и ссылки. Алголу 68 также не удалось стать широко используемым языком в основном из-за его сложности и трудностей, которые эта сложность обусловила при изучении языка и его реализации. Тем не менее некоторые из новых идей Алгола 68 в совокупности с идеями, ранее установленными в Алголе 60, вошли в базовое множество понятий проектирования практических языков и очень широко использовались.

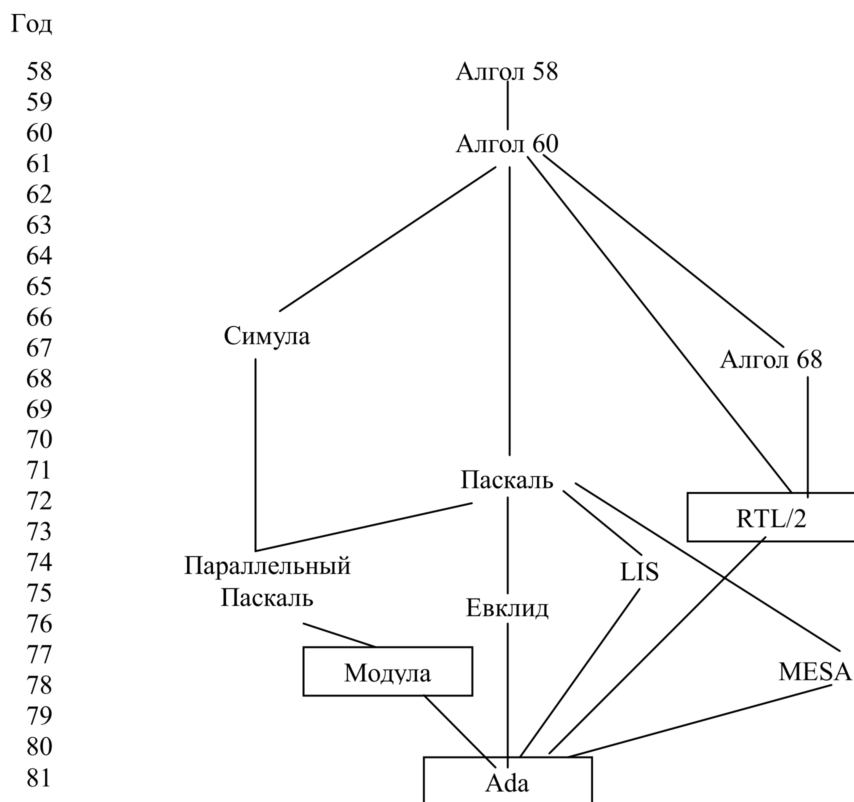


Рис. 10.1. Дерево семейства языков RTL/2, Модула и Ада.

Одним из наиболее заметных языков этого поколения является язык RTL/2 (Барнес, 1976).

Проект RTL/2 является образцом применения философии «Алголов» к проектированию языков реального времени. Это небольшой компактный и эффективный язык, предоставляющий унифицированное и надежное множество конструкций для четкого выражения программных передач управления.

Хотя RTL/2 обладает хорошими возможностями для абстракции управления, его способность абстрагировать данные рудиментарна. Второе главное направление, по которому происходило развитие, было связано с двумя новыми идеями в проектировании языков, исправившими в совокупности упомянутый недостаток. В 1970 году был изобретен язык Паскаль

(Вирт, 1971 а), в котором содержались новые идеи относительно структурирования данных. Самым главным в Паскале был принцип, согласно которому данные должны представляться в программе непосредственно в той абстрактной форме, с которой работает программист; программиста не нужно заставлять отображать абстрактные значения на небольшое число примитивных типов данных (например, целые, логические и т. д.), фактически предоставляемых базовым машинным оборудованием. Поэтому Паскаль приглашает пользователя придумывать свои собственные типы данных с помощью богатого набора базовых форм данных, таких, как перечислимые типы, записи, массивы и множества. К тому же Паскаль обеспечивает систему сильной типизации, с помощью которой гарантируется, что после определения абстрактного типа объекты этого типа будут использоваться корректно.

Хотя Паскаль ввел новые и мощные механизмы построения абстрактных типов данных, он не решал проблемы, как специфицировать множество операций, которые можно разрешить применять к этим типам. Независимо от Паскаля был разработан язык Симула (Дал и др., 1968), который содержит в качестве подмножества Алгол 60 и, что более важно, в нем введена новая конструкция программирования, называемая классом. Под классом понимается специальный вид типа данных, который позволяет специфицировать как структуру типа, так и множество операций, которые разрешается к нему применять. Однако язык Симула не располагал такими сильными типами данных, как Паскаль, что не позволило воспользоваться всеми преимуществами понятия класса. В появившемся в 1975 году языке Параллельный Паскаль (Бринч Хансен, 1975а) эти две идеи были объединены введением понятия класса в схему языка Паскаль. Последним шагом в развитии механизма абстракции данных было осознание того, что понятие класса слишком специализировано и дорого при реализации; свое выражение это получило в языке Модула (Вирт, 1977а). Основным содержанием понятия класса было то, что он предоставлял управляемые средства доступа к объектам данных, но добиться этого можно гораздо проще, ограничив «забором» область вокруг объекта и операций над ним. Все это привело к созданию конструкции модуля.

Язык Модула отметил кульминацию в революции абстракции данных, точно так же как Алгол 60 отметил кульминацию в революции абстракции управления. Конечно, было разработано много других языков, которые обладали механизмами абстракции данных. Некоторые из них, такие, как Альфард (Шоу и др., 1977) и CLU (Лисков и др., 1977), обладают очень изощренными механизмами, но ни один из них не мо-

жет похвастаться элегантной простотой модуля, имеющей место в языке Модула. И поэтому именно Модула является источником вдохновения при разработке абстракции данных в современных языках реального времени.

Хотя абстракция данных была, безусловно, основным направлением эволюции языков в течение ряда лет, появились и другие новые важные соображения. Впрочем, интересно отметить, что фактически без исключений эти новые соображения были представлены в языках, базирующихся на Паскале. Язык Евклид (Лампсон и др., 1977) был разработан с целью обеспечить существенную поддержку верификации программ при помощи внесения в текст программы утверждений. Одновременно в Евклиде были подправлены многие из недостатков, обнаруженных в Паскале. В языке LIS (Ихбиа и др., 1974) разработаны механизмы для раздельной компиляции (которая до этого представлялась хаотическим набором способов) и доведены до уровня надежной интегральной языковой возможности (Ихбиа и Ферран, 1977). Язык Mesa (Гешке и др., 1977) позволил накопить серьезный опыт использования подпрограмм обработки исключений (Гешке и Сатертвайте, 1977). Кроме того, естественно, во многих языках вводились непосредственно конструкции мультипрограммирования.

Все эти новейшие языки представляют собой эксперименты как в области проектирования языка, так и в области методологии программирования. Последним этапом этой эволюционной истории явилась публикация в 1980 году проекта языка Ада. Авторы языка Ада воспользовались лучшими идеями этих языков; в результате получился хорошо скомпонованный практический язык для встроенных систем реального времени. Ада - это мощный язык, предназначенный для программирования очень больших систем. Безусловно, он является доминирующим языком в программировании реального времени восьмидесятых годов, и во многих аспектах эта книга является описанием истории эволюции Ады.

Глава 11.
ЯЗЫК RTL/2

11.1. ПРЕДПОСЫЛКИ

Язык RTL/2 был разработан Джоном Барнесом в Центральной лаборатории прикладных исследований ICI. Реализация первой версии языка была завершена в 1971 году. В 1974 году,

после двух лет интенсивного использования в рамках SPL, он поступил на рынок (SPL, 1974). В дальнейшем он стал пользоваться большим успехом (особенно в тех странах, где не существовало правительственной политики предпочтения) и в настоящее время считается одним из основных языков реального времени. В настоящее время он находится на рассмотрении Британского института стандартов (BSI, 1979). У языка RTL/2 имеется прототип RTL/1, но этот язык был чисто экспериментальным и с появлением RTL/2 вышел из употребления (Барнес, 1972).

Первоначальная цель создания RTL/2 состояла в разработке языка, который сократил бы стоимость создания и поддержания программного обеспечения для систем реального времени и способствовал бы одновременно усовершенствованию практических методов построения надежного программного обеспечения. С этой целью был разработан язык, который базировался в основном на Алголе 60 и Алголе 68 (особенно на последнем) и содержал ряд новых собственных идей. В частности, RTL/2 предоставляет новую программную структуру, основанную на понятии модулярности одного уровня. Все программы RTL/2 строятся из набора «брусков», при этом каждый брусок обозначает либо блок данных, либо процедуру, либо стек. Все бруски обладают именами и могут вкладываться друг в друга. Поэтому с помощью брусков можно легко строить большие системы модульным способом, так как у них очень простой и четко определяемый интерфейс. Кроме того, хотя RTL/2 не поддерживает мультипрограммирование непосредственно, возможность использования стековых брусков в качестве рабочей памяти задачи позволяет довольно легко установить интерфейс языка с многозадачной операционной системой.

В следующем разделе язык RTL/2 описывается подробнее. Затем в разд. 11.3 следует пример использования языка. В заключение в разд. 11.4 содержится обсуждение и оценка языка. Те, кто хочет изучить язык или глубже понять заложенные в нем принципы, могут обратиться непосредственно к публикации Барнеса (1976).

11.2. ЯЗЫК

Хотя язык RTL/2 ориентирован специально на программирование в реальном времени, специфических особенностей реального времени в нем на самом деле не имеется. Например, нет встроенных возможностей для фиксации времени, многозадачного прогона или ввода-вывода. Их исключение из языка объясняется тем, что за счет этого становится возможным

пропускать написанные на RTL/2 программы на очень маленьких машинах, не отказываясь от возможностей, которые могут предоставить очень большие машины. В первом случае минимальные возможности могут предоставляться небольшим ядром операционной системы, требующей менее одного килобайта памяти, тогда как во втором случае развитые возможности могут предоставляться большой операционной системой общего назначения с базирующейся на дисках файловой памятью. Конечно, у этой стратегии имеется тот существенный недостаток, что может возникнуть серьезная проблема, связанная с мобильностью написанных на RTL/2 программ. Поэтому на практике были разработаны некоторые соглашения. Для ввода-вывода и обработки ошибок были установлены строгие стандарты, которых придерживалось большинство реализаций. Впрочем, в области многозадачности следовало ожидать большего разнообразия ситуаций и твердых стандартов установлено не было.

В соответствии с этим излагаемый здесь язык RTL/2 не содержит никаких возможностей мультипрограммирования. В разд. 11.2.1 описывается стандартный язык, и в разд. 11.2.2 кратко даются стандарты ввода-вывода и обработки ошибок. Впрочем, чтобы показать, как RTL/2 может использоваться при построении многозадачных систем, в разд. 11.3 приводится в общих чертах пример простого многозадачного исполняющего ядра, написанного на RTL/2.

И еще следует заметить, что во всех дальнейших примерах программ используются те же самые типографские условности, что и во всей этой книге, т. е. идентификаторы выделяются курсивом, а зарезервированные слова - жирным шрифтом. Это немного не соответствует действительности, так как в RTL/2 предопределенные типы также являются зарезервированными словами. Поэтому, если мы хотим быть точными, переменная предопределенного типа *целый* должна быть описана так: **int** *j*, тогда как переменная определенного пользователем типа *некоторый тип* должна быть описана так: *некоторый тип **j***. Чтобы сохранить в книге единую систему написания, предопределенные типы будем записывать как обычные идентификаторы.

11.2.1. Формальный язык

В RTL/2 имеется четыре простых типа данных, все числовые. Это следующие типы: *байтовый, целый, дробный* и *вещественный.* Но RTL/2 является языком со слабой типизацией, что позволяет использовать эти числовые типы для самых различных целей. Поэтому, хотя в языке нет логических, символь-

ных и перечислимых типов, их можно полностью компенсировать, используя числовые типы.

Байтовый тип используется для представления чисел в диапазоне от 0 до 255; предполагается, что он будет отображаться на 8-байтовую память базового машинного оборудования. Его можно также использовать для представления битовых строк длины 8 и символьных типов. Тип *целый* используется для представления целых со знаком с предполагаемым минимальным диапазоном от —32768 до 32767. Его можно также использовать для представления битовых строк фиксированной длины и целых частей чисел с фиксированной запятой. *Дробный* тип используется для представления целых со знаком в диапазоне от -1 до +1 (включая -1, но не +1). Он имеет то же самое базовое представление, что и целый тип, с тем исключением, что предполагается, что двоичная точка расположена в крайнем положении слева, а не справа. И последнее, *вещественный* тип используется для представления чисел с плавающей запятой, причем точность и диапазон определяются каждый раз при реализации.

Для описания переменных пишут имя типа, за которым следуют имена переменных в виде списка. Например, текст

вещественный x, y;
целый i, j;
дробный f;
байтовый b;

описывает переменные *x, y* типа *вещественный*; *i, j* типа *целый*; *f* типа *дробный* и *b* типа *байтовый.* При описании переменной ей может быть присвоено начальное значение следующим образом:

вещественный x:=1.0;

здесь *x* присваивается начальное значение 1.0.

Прежде чем описать имеющиеся операции для числовых типов, необходимо отметить, что RTL/2 автоматически преобразует один тип в другой, так чтобы он соответствовал ожидаемому типу операндов. Например, операция + не определена для типа *байтовый.* Тем не менее запись *b* + 1 корректна. Компилятор преобразует *b* в целый тип и затем прибавит 1, результат будет целым. Общий принцип такого вынужденного преобразования (ср. терминологию Алгола 68) требует, чтобы такое преобразование применялось тогда лишь, когда не происходит потери точности. В остальных случаях преобразование должно быть указано ясно, с использованием имени типа, как одноместная операция. Например, в выражении *x* := *i* *i* автоматически преобразуется (т. е. вынужденно преобразуется) в

вещественный тип, но в *i*:= **int** *x* значение *x* должно быть явно преобразовано в целый тип с помощью операции **int,** так как при преобразовании может произойти потеря точности. В RTL/2 выполняются следующие вынужденные преобразования:

(*a*) *байтовый -> целый -> вещественный*
(*b*) *дробный -> вещественный*

Однако на самом деле их имеется значительно больше, так как RTL/2 определяет четыре временных типа двойной длины для работы с дробями: *большой целый, точный дробный, точный целый* и *большой дробный.* Схема арифметики RTL/2

Таблица 11.1. **Одноместные операции (упрощенно)**

Операция	Типы операндов	Действие
+	любой	идентичность
-	*целый, дробный, вещественный*	отрицание
abs	любой	абсолютное значение
not	*целый*	дополнить каждый бит
real	любой	
int frac byte	*целый, вещественный дробный,* *вещественный байтовый, целый,* *вещественный*	преобразования типа

для фиксированной запятой по существу совпадает со схемой, описанной в разд. 2.4.3, и здесь мы ее повторять не будем. Впрочем, следует отметить, что введение дробного типа и всех операций и вынужденных преобразований, которые необходимы для его поддержания, по существу как бы удваивает сложность базовой системы типизации.

Так как в RTL/2 целые и байтовые величины выступают в различных ролях и из-за сложности работы с дробями, необходимо достаточно большое множество операций. В упрощенной форме они приведены в табл. 11.1 и 11.2. Таблица 11.1 для типа *фиксированный* справедлива и для типов *целый, дробный* и форм с двойной длиной. Таблица упрощена, в каждом случае разрешаются не все фиксированные типы.

Из этих таблиц можно видеть, что вещественная арифметика поддерживается обычным способом с помощью одноместных операций +, -, abs и двуместных операций +, —, *, /. Целая и дробная арифметики несколько более сложные. Аддитивные операции + и — применяются к операндам *целый* или *дробный* одинарной длины, давая в результате тот же самый тип. Операция умножения применяется к операндам одинарной длины, давая при выполнении результат двойной длины. Две

Таблица 11.2. **Двуместные операции (упрощенно)**

Оператор	Тип 1-го операнда	Тип 2-го операнда	Тип результата	Приоритет	Действие
sll, **srl**, **shl**	*целый*	*целый*	*целый*	6	Логические сдвиги
sla, **sra**, **sha**	*фикс*	*целый*	*фикс*	6	Арифметические сдвиги двойной длины
*	*фикс, вещ*	*фикс, вещ*	*фикс, вещ*	5	Умножение
:/	*фикс*	*фикс*	*целый*	5	Фиксированное деление – целый результат
//	*фикс*	*фикс*	*дробный*	5	Фиксированное деление – дробный результат
/	*вещ*	*вещ*	*вещ*	5	Вещественное деление
mod	*фикс*	*фикс*	*фикс*	5	Модуль
land	*целый, байт*	*целый, байт*	*целый, байт*	4	Битовое логическое и
lor	*целый, байт*	*целый, байт*	*целый, байт*	3	Битовое логическое или
nev	*целый, байт*	*целый, байт*	*целый, байт*	2	Битовое логическое или – искл.
+	*фикс, вещ*	*фикс, вещ*	*фикс, вещ*	1	Сложение
-	*фикс, вещ*	*фикс, вещ*	*фикс, вещ*	1	Вычитание

операции деления применяются к делимому двойной длины и делителю одинарной длины. Операция :/ дает *целый* результат, а операция // дает *дробный* результат, например

5:/2 = 2,   1//2 = 0.5

Арифметические операции сдвига **sla**, **sra** и **sha** используются главным образом для масштабирования и применяются к операндам двойной длины. Например, в операторе

i ;=(*i***j*) **sra** 16

результат *i***j* является целым двойной длины, который сдвигается на 16 позиций вправо до отбрасывания слова высокого порядка и преобразуется обратно в целый тип одинарной длины, т. е. сдвиг происходит до того, как *i***j* преобразуется к одинарной длине.

Логические операции **not**, **land**, **lor**, **nev** и операции сдвига **sll**, **srl**, **shl** обеспечивают работу с битами для целых операндов (**land**, **lor** и **nev** применимы также и к типу *байтовый*). Например, предположим, что два байтовых значения 61 и 62 нужно упаковать в одно целое *i*. Это можно сделать следующим образом:

i:=(*b*1 **sll** 8) **lor** *b*2

Заметим, что **sll** применяется только к целым типам, поэтому *b*1 сначала вынужденно преобразуется в целый, а затем сдвигается влево на 8 позиций. Из этого в свою очередь следует, что значение *b*2 должно быть вынужденно преобразовано также в целый тип перед участием в операции «или» со сдвинутой версией *b*1.

Структурирование данных осуществляется в RTL/2 с помощью массивов, записей и ссылочных типов. Ссылочный тип это по существу указательный тип, он используется очень широко. Наряду с обеспечением возможности построения сложных структур данных он обеспечивает также единственное средство возвращения результатов из процедур. Для введения ссылочного типа необходимо в начале написать ключевое слово **ref**. Например, во фрагменте

целый i, j;
ref *целый ri* := *i,*

переменная *ri* содержит ссылку на целую переменную *i* в связи с инициализацией *«ri* := *i*». Впрочем, она может содержать ссылку на любую целую переменную, поэтому вполне законно впоследствии написать *ri*:=*j,* так чтобы *ri* содержала ссылку на *j*. Язык RTL/2 обеспечивает автоматическую расшифровку ссылки, т. е. *ri* :=2 означает, что значение 2 присваивается переменной, на которую ссылается *ri* (т. е. *j*), а не самой *ri.* Расшифровку переменной можно также выполнить с помощью операции **val**, т. е.

val *ri*:=*i*

означает «*j* :=i», а не *«ri* теперь ссылается на *i».*

Массивы в RTL/2 могут иметь любое число индексов, но их нижняя граница всегда равна 1. Типом компонент может быть любой скалярный тип или комбинированный тип, но не другой регулярный тип. Например,

array(31) *вещественный ежедневныеосадки*;

описывает *вещественный* массив, содержащий 31 элемент, так что *ежедневныеосадки* (1) означает его первый элемент. Многомерные массивы всегда представляются массивами ссылок на массивы. Например, рассмотрим

array(3,3) *байтовый шахматнаядоска*; что фактически

представляется в виде

array (3) *байтовый r*1,*r*2, *r*3;
array(3) **ref array** *байтовый шахматнаядоска* :=(*r*1,*r*2, *r*3);

где *шахматнаядоска* теперь представлена в виде одномерного массива ссылок, начальные значения которого указывают на три одномерных массива байтов, называемых *r*1*, r*2*, r*3. Заметим, что (*r*1*, r*2*, r*3) является конструктором массива. Конечно, в первом случае имена строк анонимны.

Записи определяются в RTL/2 с помощью определения *вида* (вид является синонимом типа в терминологии Алгола 68). Например,

mode *данныешина* (*вещественный ширина, диаметр, целый здав, пдав*);

описывает тип записи с двумя вещественными компонентами, называемыми *ширина* и *диаметр,* и двумя целыми компонентами, называемыми *здав* и *пдав.* Как правило, компоненты записи могут быть любого типа, за исключением других записей или массивов записей. Переменные типа запись описываются обычным способом, например,

данныешина форд, фиат;

Доступ к компонентам записи осуществляется с помощью традиционного употребления точки для этих целей; например,

форд.пдав :=22;

установит переднее давление шины со значением 22. Другим способом доступа к компонентам записи является использование конструктора записи, например так:

фиат :=(12.5, 144.0, 21, 24)

Наконец, заметим, что вид определения записи является единственной возможностью разрешенного в RTL/2 определения типа. Поэтому нельзя составить описание

mode *вектор* **array**(100) *вещественный*

и затем описать переменные типа *вектор* в виде, например,

вектор мойвектор;

Впрочем, аналогичный эффект может быть получен с помощью определения **let** (пусть). Это соответствует возможности простого замещения текста. Тогда текстовой строке присваивается имя. Поэтому вид требуемого вектора может быть введен с помощью

let *вектор* = **array**(100) *вещественный*;

в этом случае описание переменной в форме *вектор мойвектор*; текстуально эквивалентно записи

```
array(100) вещественный мойвектор;
```

Определения **let** широко используют для введения констант. Язык RTL/2 обеспечивает три формы структурного оператора управления. Условные операторы имеют следующий общий вид:

```
if условие1 then S1
elseif условие2 then S2
elseif условие3 then S3
     . . . . . . . .
else S0
end;
```

где *Si* обозначает произвольную последовательность операторов. Части **elseif и else** необязательны, может содержаться произвольное количество частей **elseif.** Так как в RTL/2 нет логического типа, условия должны выражаться в форме выражений отношения. Операции отношения = и # (не равно) применимы к любым численным типам RTL/2, а также к типам *метка, стек* и *процедура* (которые описаны ниже). Кроме этого, операции <, >, <=, >= могут применяться к целым, дробным и вещественным типам. Так как в RTL/2 имеется ссылочный тип, необходимы еще две дополнительные операции отношения. Операция :=: проверяет, ссылаются ли две ссылочные переменные на один и тот же объект или нет, а :#: является обратной операцией. Такой же синтаксис, как в приведенной выше форме условных операторов, может быть использован для обозначения условных выражений. Например,

```
i :=if j=0 then -1 else 1 end;
```

присвоит *i* значение -1, если *j* = 0, и 1 в противном случае.

Итерация обеспечивается операторами **for** и **while,** которые имеют следующий вид

```
for индекс :=e1 to e2 by e3 do
  B
rep
```

и

```
while условие do
  S
rep
```

Значение части **by** в цикле **for** может быть как положительным, так и отрицательным, оно может быть опущено - значение по умолчанию равно 1. Переменная управления *индекс*

строго локализована в цикле. Если явно ссылаться на эту переменную не нужно и приращение равно 1, то разрешается такая сокращенная форма:

```
to n do
  B
rep
```

что означает итерацию тела цикла *n* раз.

Язык RTL/2 обладает блочной структурой. Все переменные, которые используются в программном модуле, должны быть явно описаны либо глобально в бруске данных, либо локально в блоке. Блок можно ввести в любом месте программы, где можно написать оператор. Его скобками являются ключевые слова **block** и **endblock;** применяются обычные правила области действия. Например, в следующем фрагменте программы вещественная переменная *x*, описанная во внешнем блоке, недоступна во внутреннем блоке, но вещественная переменная *y* доступна в обоих блоках:

```
block
  вещественный x, y;
    …
  block
    целый x;
      ... % целый x, вещественный y здесь доступны %
      ... % целый x, вещественный y здесь доступны %

endblock;
```

Локально в блоке могут быть описаны только простые типы. Структурные переменные должны описываться в бруске данных.

Вернемся к операторам итерации. Тело цикла **for** является фактически блоком, в котором локально описана индексная переменная. Поэтому переменные можно описывать локально в цикле **for,** как, например, в следующем примере

```
for i := 1 to length a do
  ref вещественный r := a(i);
  val r := r + 1.0/r;
rep
```

где локальная ссылочная переменная *r* введена для того, чтобы вычисление адреса, обеспечивающее доступ к компоненте вещественного массива *a,* выполнялось только один раз при выполнении тела цикла. Заметим, что **length** является одноместной операцией, вычисляющей длину своего операнда (в нашем

случае массива *a*). Однако тело цикла **while** является не блоком, а простой последовательностью операторов.

Кроме этих структурных операторов управления язык RTL/2 разрешает также программировать и непосредственные переходы. Каждый оператор может быть помечен меткой, перед оператором пишется идентификатор, за которым следует двоеточие, например,

$L1 : i := i + 1;$

есть оператор, помеченный литеральной меткой *L*1. Для программирования перехода на эту метку можно использовать либо оператор **goto**

goto *L*1

либо оператор **switch,** как, например,

switch *k* **of** *L*1, *L*2. *L*3 **...** *LN*

где передача управления будет происходить на метку *L*1, если $k = 1$, на метку *L*2, если $k = 2$ и т. д. Если *k* меньше 1 или больше *N,* то оператор **switch** просто не принимается во внимание.

Оператор **goto** предназначен главным образом для обработки ошибок и поэтому является более общим средством, чем оператор **switch.** Специфицируемые в операторе **switch** метки должны быть литеральными метками. Оператор же **goto** может ссылаться на любое выражение, которое в качестве результата вырабатывает метку. В частности, он может ссылаться на меточную переменную. Меточные переменные описываются следующим образом:

label *l*;

Им могут присваиваться значения в обычном операторе присваивания, как, например $l := L1$. Поэтому когда выполняется оператор **goto**, такой, как goto *l*, место передачи управления зависит от текущего значения *l*. Оператор **goto** не локализирован в процедуре, поэтому он может использоваться для передачи управления вовне из вложенной последовательности обращений к процедурам. Таким образом, в сочетании с меточными переменными он является мощным механизмом при использовании для обработки ошибок (см. разд. 8.2).

Остальные особенности языка RTL/2, которые нам нужно рассмотреть, относятся к общей структуре программы. Программа на RTL/2 состоит из одного или более отдельно компилируемых модулей. Каждый модуль строится из брусков трех типов: *стековые* бруски, бруски *данных* и *процедурные* бруски.

Стековой брусок описывает область памяти, которая используется задачей в качестве рабочей памяти. Например,

```
stack сканнер 1000;
```

описывает стековый брусок, называемый *сканнером,* который содержит 1000 единиц памяти (размер единицы зависит от реализации). Нет встроенных языковых особенностей для связывания процедуры со стеком (см. разд. 11.2.3), но можно описать переменные типа *стек,* что позволяет ссылаться на задачи. Например, описание

```
stack s := сканнер;
```

вводит стековую переменную *s* и присваивает ей начальное значение для обозначения стека *сканнер.*

Бруски данных используются для представления статических областей памяти, содержащих простые, регулярные и комбинированные типы. Например, описание

```
data локальный;
  целый i, j;
  array(6) целый уровень;
enddata;
```

определяет блок данных, называемый *локальный,* содержащий целые переменные *i* и *j* и регулярную переменную *уровень.* На описываемые в бруске данных переменные можно непосредственно ссылаться в любом месте объемлющего модуля (если только их имена не спрятаны в описаниях локального бруска). К ним можно также обращаться из других модулей так, как описано ниже.

Процедурные бруски обозначают последовательности действий, которые необходимо выполнить. Рассмотрим в качестве примера следующую процедуру сложения двух векторов *a* и *b,* вычисляющую в результате вектор *c*:

```
proc сложвектор(ref array вещественный a, b, c);
   for i := 1 to length a do
      c(t):=a(i) + b(i);
   rep
endproc
```

Параметрами процедуры могут быть только простые типы, поэтому для передачи в процедуру массива должны использоваться ссылочные переменные. Кроме того, передача параметров всегда осуществляется вызовом по значению, поэтому и здесь должна использоваться ссылочная переменная, если мы хотим, чтобы процедура вырабатывала результат. Данная процедура

сложвектор может быть вызвана с помощью оператора обращения к процедуре, например

сложвектор(*v*1, *v*2, *v*3);

где *v*1, *v*2, *v*3 - вещественные массивы. Отметим, что фактическими параметрами могут быть любые выражения, вырабатывающие значение требуемого типа. В нашем случае имя какого-либо вещественного массива порождает значение **ref array** *вещественный.*

Процедуру можно также использовать для возврата значения в качестве функции. Например,

> **proc** *макс*(*целый i, j*) *целый*;
>   **return** (**if** *i* > *j* **then** *i* **else** *j* **end**);
> **endproc**

определяет функцию *макс,* которую далее можно использовать в качестве компоненты выражения, например

i :=*макс*(*j,* 0) + 1;

Тип возвращаемого значения должен указываться после списка параметров, а фактическое значение возвращается в теле процедуры оператором **return.** Оператор **return** можно также использовать (с параметром или без него) для преждевременного завершения процедуры. Например,

> **proc** *копировать*(**ref array** *целый откуда, куда*);
> **if** *откуда* :=: *куда* **then return end;**
>   **…**
>   *% копировать откуда в куда %*
>   **…**
> **endproc**

Телом процедурного бруска является блок, поэтому перед последовательностью операторов могут находиться описания локальных переменных. Впрочем, как и в случае простых блоков, они ограничиваются только простыми типами.

Как и применительно к стекам, для передачи имен процедур другим процедурам предоставляются процедурные переменные. Описание процедурной переменной задает число и тип параметров, но не их имена, например

proc(*целый, целый*) *целый f* :=*макс*;

описывает процедурную переменную *f,* которая может указывать на процедуры с двумя входными целыми параметрами и возвращаемым целым результатом; начальное значение

переменной указывает на процедуру *макс.* Расшифровка процедурных переменных выполняется автоматически, поэтому

$i := f(j, 0) + 1;$

вызовет выполнение действий, аналогичных действиям при обращении к процедуре *макс* в предыдущем примере.

Бруски могут описываться в модуле в произвольном порядке. В качестве примера структуры модуля рассмотрим модуль, реализующий целый стек.

```
option(1) BC;    % Позволяет проверять границы массива %
title целый стек;    % Заголовок для документации %
let размер = 100;
mode стекзаписей = (array(размер) целый стек, целый верх);
data локальный;
   стекзаписей s;
   метка ошибка;
enddata;
ent proc нач(ref метка e);
   ошибка :=e;
   s.верх :== 0;
endproc;
ent proc втолкнуть(целый x);
   if s.верх = размер then goto ошибка end;
   s.верх := s.верх + 1;
   s.стек(s.верх) := x;
endproc;
ent proc вытолкнуть(ref целый x);
   s.верх := s.еерх - 1;
   if s.верх = 0 then goto ошибка end;
   val x := s.стек(s.верх);
endproc;
```

Реальный стек представляется записью, содержащей массив для хранящихся в стеке данных и стековый указатель. Переменная *s* типа *стекзаписей* описывается в бруске данных *локальный,* она доступна только через процедуры *нач, втолкнуть* и *вытолкнуть.* К этим процедурам можно производить обращения из других модулей, так как перед их описанием стоит префикс **ent**. Для использования нашего стека в другом модуле он должен быть описан во внешней процедуре. Например

```
ext проц(ref метка) нач;
ext проц(целый)втолкнуть;
ext проц(ref целый)вытолкнуть;
proc взятьстек( );
   int i;
      …
     нач(ош);
     втолкнуть(1);
     втолкнуть(2);
     вытолкнуть(i);
     …
       и т. д.
   ош: ... % здесь переход при ошибке стека %
endproc
```

Бруски данных можно сделать доступными извне точно так же, как и процедуры, с помощью предшествующего им ключевого слова **ent.** Внешний брусок данных специфицируется ключевым словом **ext.** Например, описание

ent data *глобальный*;
целый i;
вещественный x;
enddata

может стать доступным из другого модуля, если в тот модуль включено описание

ext data *глобальный*;
целый i;
вещественный x;
enddata;

Внешние бруски данных выполняют роль, аналогичную именованным COMMON в Фортране, но они более надежны в том отношении, что имена и типы составляющих переменных проверяются во время связывания.

11.2.2. Стандартный ввод-вывод и восстановление после ошибок

Язык RTL/2 предоставляет специальную форму внешней спецификации для тех случаев, когда требуется связывание не с некоторым другим модулем, а с операционной системой. Она имеет такую же форму, как и обычное внешнее описание, но ключевое слово **ext** заменяется на **svc** (связь с супервизором). Кроме того, в случае брусков данных **svc** для каждой задачи поддерживается приватная копия бруска в отличие от ситуаций, когда все задачи обращаются к одному центральному бруску.

Стандартный ввод-ввод определяется только для потоков ввода-вывода. Брусок данных **svc** с именем *пвв* определяется следующим образом:

svc data *пвв*;
proc( )*байт ввод*;
proc(*байт*)*вывод*;
enddata;

Этот брусок содержит две процедурные переменные, *ввод* служит для ввода одного байта из текущего вводного потока, а *вывод* — для вывода одного байта в текущий выводной поток. Переключение потоков осуществляется простым присваиванием имен процедур управления физическими устройствами переменным *ввод* и *вывод*. Все подпрограммы ввода-вывода более высокого уровня используют *ввод* и *вывод как* примитивы, поэтому обеспечивается простая иерархическая схема ввода-вывода. Пользователю очень просто написать свою собственную подпрограмму управления для конкретного устройства (например, используя вставки на коде ассемблера), и затем воспользоваться преимуществами подпрограмм высокого уровня, присваивая имена своих собственных подпрограмм переменным *ввод* или *вывод* подходящим образом.

Предоставляемые процедуры ввода-вывода высокого уровня позволяют читать из текущих потоков ввода-вывода и писать в них текстовые строки, дробные, целые и вещественные числа. Имеются два варианта каждой из подпрограмм числового вывода, предоставляющие форматную и неформатную распечатку. Например, *пст*(19) непосредственно выдает два символа '!' и '9', тогда как форматный вариант *пстф*(*19, f*) выдаст число 19 в поле длиной *f* символов.

Для того чтобы использовать процедуры чтения, необходимо описать еще один брусок данных **svc.**

```
svc data чтт;
   байт заклс;
   байт флагвв;
enddata;
```

Переменной *заклс* всегда присваивается заключительный символ, который встречается в вводном потоке вслед за операцией чтения. Например, если к *читать* (*i*) обращаются для чтения в *t* целого числа и в вводном потоке появляется последовательность символов 100,..., то в *i* будет заслано значение 100, а в *заклс* будет помещен символ ','. *Флагвв* используется для регистрации любой ошибки, которая происходит при вводе, перед обращением к процедуре обработки исправимых ошибок *ошп*$_{г}$ описываемой ниже.

Стандарты восстановления после ошибок различают два вида ошибок: исправимые и неисправимые. Обработка ошибок использует следующий брусок данных **svc**:

```
svc data ош;
   label ошм;
   целый ошн;
   proc(целый)ошп;
enddata;
```

При возникновении исправимой ошибки (например, во время исполнения подпрограмм чтения) вызывается процедурная переменная *ошп* с соответствующим кодом ошибки в качестве аргумента. Поэтому пользователь может присвоить эту переменную одной из своих собственных процедур восстановления после ошибок для принятия соответствующих мер. При возникновении неисправимой ошибки (например, нарушение границ массива) код ошибки помещается в *ошн* и происходит передача управления на *ошм.* И здесь пользователь может присвоить подходящее значение *ошм,* чтобы вызвать обращение к одной из своих собственных подпрограмм восстановления. Различие между двумя методами обработки ошибок заключается в том, что при обращении к процедуре обработки ошибок естественный поток передач управления не нарушается, тогда как в случае неисправимых ошибок переход к *ошм,* как правило, вызовет завершение активных в этот момент процедур. Конечно, основным здесь является то, что использование меток и процедурных переменных обеспечивает для пользователя гибкость при программировании своих собственных операций восстановления.

11.3. ИСПОЛЬЗОВАНИЕ ЯЗЫКА RTL/2

Хотя в языке RTL/2 не имеется встроенных возможностей мультипрограммирования, язык разработан так, чтобы его можно было легко ввести в многозадачную систему. Двумя ключевыми возможностями языка, которые облегчают такую интеграцию, являются стековый брусок и явный механизм раздельной компиляции. В этом разделе такое использование RTL/2 будет показано на примере разработки в RTL/2 простого ядра многозадачного супервизора. Этот супервизор обеспечивает составления расписания для фиксированного числа задач одинакового приоритета на один физический процессор. Задачи пользователя имеют возможность запускать и останавливать задачи, вступать в связь между собой с помощью семафоров и сигналов, задерживать выполнение задач на фиксированный период. Практический пример такого рода системы читатель может найти в серии операционных систем MTS, которые были специально разработаны для поддержания программ, написанных на RTL/2 (ICI, 1973).

В нашем примере интерфейс между ядром и задачами пользователя будет осуществляться с помощью внешнего бруска данных, множества внешних процедур и множества определений

let, задающих системные параметры. Такими параметрами обычно бывают следующие:

```
let чзадач=10;      % Макс число задач %
let чсиг = 20;      % Макс число сигналов %
let чсем = 25;      % Макс число семафоров %
let активн = 0;     % Код статуса для активной задачи %
let останов = 1;    % Код статуса для прерванной задачи %
```

Используя эти определения, пользователь должен затем задать брусок данных, описывая имена стека и главной процедуры для каждой задачи и ее начальный статус, например

```
ext проц( )главная1, главная2, ...;
ext стек стк1, стк2, ...;
ent data записизадач;
   array(чзадач)стекзстек :=(стк1, стк2, ...);
   array(чзадач)проц( )зпроц := (главная 1, главная2,...);
   array(чзадач)целый зстатус := (актив, останов,...);
enddata;
```

Поэтому структура задачи пользователя будет иметь, например, такой вид:

```
option(l);
title задача пользователя1;
% Стек рабочей памяти задачи %
ent stack стк1 1000;
% Процедуры ядра %
ext proc(целый)старт, стоп, задержать, послать, ждать,
                                    оградить, освободить;
% Основное тело задачи %
ent proc главная1( );
   ...
   … % Код для задачи1 %
endproc;
% Другие бруски для этой задачи %
...
```

На каждую задачу, сигнал и семафор ссылаются с помощью уникального целого идентификатора в диапазоне, специфицированном в определениях **let.** Обращения к процедурам

старт(*T*) *и стоп*(*T*)

может использовать любая задача для запуска или остановки задачи *T*. Обращения

послать(*N*) *и ждать*(*N*)

пошлют или будут ждать сигнал *N* соответственно. Обращения

оградить(*S*) *и освободить*(*S*)

закроют или освободят семафор *S* соответственно. Семантика последних четырех операций определена в разд. 6.2. И наконец, обращение

задержать(*K*)

приведет к задержке выполнения задачи на *K* единиц времени.

Основная функция ядра состоит в распределении ресурсов физического процессора среди всех задач пользователя. Примем простую стратегию составления расписания «по кускам», когда контекстное переключение происходит каждый раз при получении прерывания от часов реального времени. Кроме этого, контекстное переключение будет происходить также при обращении к любой из процедур ядра. Очевидно, что подробный код физического контекстного переключения нельзя выразить на языке RTL/2, так как это требует непосредственной манипуляции с регистрами процессора. В реальной системе это будет сделано с помощью вставок кода на языке ассемблера; в нашем примере такие действия будут указываться в комментариях.

Фактическое контекстное переключение будет осуществляться двумя процедурами *войти* и *покинуть.* К этим процедурам обращаются при входе в ядро и при выходе из ядра. Процедура *войти* сначала просто выключает прерывания, а затем записывает регистры процессора для текущей задачи на стек прерывания. Непосредственное контекстное переключение выполняется процедурой *покинуть,* которая выбирает новую задачу для прогона, переключает контекст со старой задачи на новую и затем восстанавливает прерывания. Основное содержание ядра, таким образом, описывается следующими брусками:

```
% Системные определения let   %
let чзадач=10;                 % Макс число задач %
let чсиг = 20;                 % Макс число сигналов %
let чсем = 25;                 % Макс число семафоров %
let активн = 0;                % Коды статуса задачи %
let останов = 1;               % Статус = время задержки %
let ждущая = 2;
% Определенные пользователем записи задачи %
ext data записизадачи;
```

```
  array(чзадач)стек зстек;
  array (чзадач)проц( ) зпроц( );
  array (чзадач)целый зстатус;
enddata;
% Работающая задача %
data идрабзад;
  целый зработающий;
enddata;
% Стек ядра %
stack ядро;

   % Программы входа-выхода ядра %
proc войти( )
   % Выключить прерывания %
   % Записать регистры, процессора в стек текущей задачи %
   %! Базовый адрес возврата! %
endproc;
proc покинуть( );
  целый новаязадача;
  новаязадача := выбрать( );
  % Запомнить текущий указатель стека процессора в базе   %
  % старых стеков где                                     %
  % старые = if работающая = 0 then ядро else зстек       %
  % (зработающая) end                                     %
  % Загрузить указатель стека процессора из базы новых    %
  % где                                                   %
  % новые = if новаязадача = 0 then ядро else зстек       %
  % (новаязадача)                                         %
  % Загрузить счетчик программ процессора меткой выход    %
  выход : зработающая := новаязадача;
  % Восстановить регистры процессора из текущего стека    %
  % (Замечание: это будет под адресом возврата            %
  % Восстановить прерывания %
endproc;
```

В этом примере процедура *выбрать* возвращает номер следующей программы для ее прогона; если возвращается 0, то это означает, что активных задач нет и выполняется задача ядра. Эта последняя задача активизируется в самом начале работы системы. Это единственная дополнительная процедура в системе, которая требует кодирования на машинном уровне.

```
 proc иницядро( );
  for i:=1 to чзадач do
    % Запомнить начальные регистры процессора в зстеке(i)    %
    % Запомнить счетчик начальной процессорной               %
    % программы, в зстеке(i) зпроц(i)                        %
    % Заслать указатель стека в базу зстека(i)               %
    % для указания на вершину стека                          %
  rep;
  иницочереди;
  войти( ); покинуть( );
  цикл ; goto цикл
endproc;
```

К процедуре *иницядро* обращается системный загрузчик со стеком ядра, который инициализирован соответствующим образом; процедура инициализирует стеки задач каждого пользователя и очередь задач (описано ниже) и обращается к программам *войти* и *покинуть.* Непосредственно перед обращением к программе *войти* различные задачи будут находиться в состоянии, которое показано на рис. 11.1(a), где предполагается, что механизм процедурного обращения запоминает адрес возврата на вершине стека. В результате последовательности обращений к *войти* и *покинуть* возникнет ситуация, показанная на рисунке 11.1(b), где предполагается, что подпрограмма *выбрать* выдает в качестве следующей рабочей программы задачу 1. Начиная с этого момента результатом всех входов в ядро будет аналогичная последовательность событий, когда все обращения к *войти* и *покинуть* приведут к требуемому контекстному переключению. Когда происходит повторное обращение к ядру (так как нет активных пользовательских задач), ядро просто выполняет пустой цикл до тех пор, пока не станет активной некоторая задача пользователя.

Остальная часть системы может быть теперь описана полностью на языке RTL/2. Каждый раз при получении прерывания от часов реального времени выполняются регулярные контекстные переключения. Будем предполагать, что эффект такого прерывания заключается в простом обращении к следящей за часами процедуре с помещением в стеке прерванной задачи значения программного счетчика в точке прерывания (т. е. его эффект идентичен обращению к обычной процедуре),

```
proc часы( );
 войти( );
  for i := 1 to чзодач do
  ref целый статус := зстатус(i);
  if статус < 0 then
     статус := статус + 1
  end;
 rep;
  покинуть( );
endproc;
```

Каждое обращение *к* процедуре *часы* вызывает увеличение на единицу значения *статуса* для всех задач, находящихся в этот момент в состоянии задержки, и контекстное переключение в связи с обращением к *войти/покинуть.*

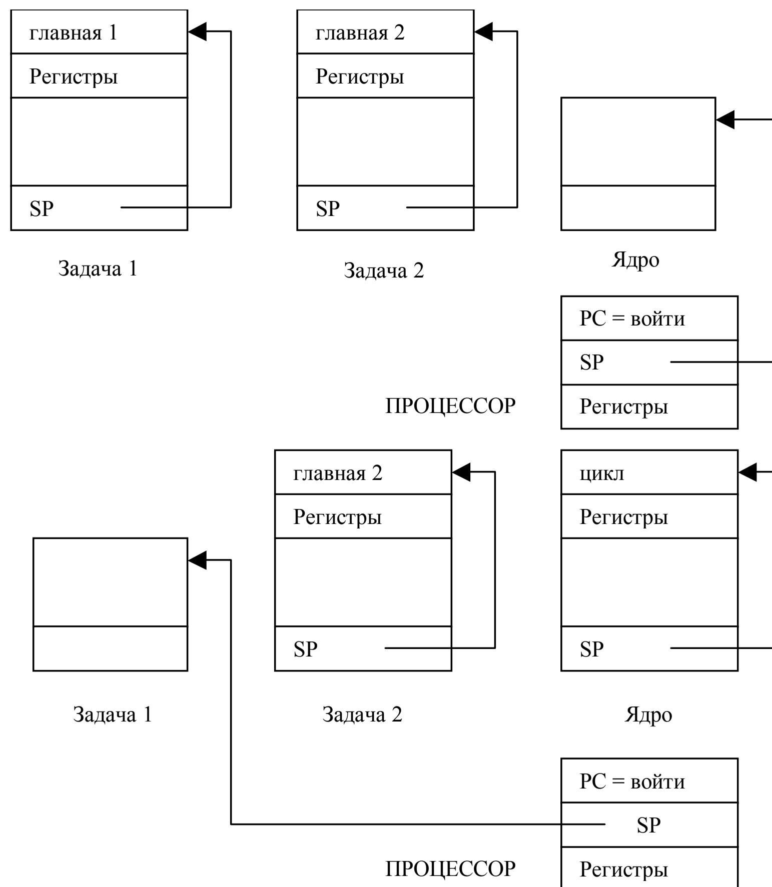


Рис. 11.1. Начальное состояние стеков, (а) Начальное состояние перед обращением *«иницядро»* к «войти». (b) Состояние при возвращении из «покинуть».

Для нашего простого супервизорного ядра возьмем простой алгоритм составления расписания на основе последовательной замены. Он реализуется с помощью процедуры *выбрать,* которая проверяет записи задач и выбирает следующую задачу с активным статусом. Если такой не существует, процедура воз-

вращает значение 0, что приводит к выполнению пустого цикла до следующего прерывания от часов.

```
% Алгоритм расписания %
proc выбрать( )цел;
    целый зид := зработающая;
    целый зсчетчик :— чзадач;
      байтовый пустой := 0;
      while зсчетчик > 0 and пустой = 0 do
         зид :=(зид mod чзадач) + 1;
         зсчетчик := зсчетчик - 1;
         if статус(зид) = активный then
            пустой :== 1;
         end;
    rep;
  return(if пустой = 0 then 0 else зид end);
 endproc;
```

Остальная часть ядра относится к реализации системных процедур. Процедуры *стоп, старт* и *задержать* тривиальны, их можно привести без обсуждения.

```
% Подпрограммы старт/стоп %
ent proc стоп(целый идзадача);
  войти( );
  зстатус(идзадача) := останов;
  покинуть( );
endproc;
ent proc старт(целый идзадача);
  войти( );
  зстатус(идзадача) := активный;
  покинуть( );
endproc;
% Подпрограмма задержки %
ent proc задержать(целый время);
  войти( );
  зстатус(зрабочий) := - время;
  покинуть( );
endproc;
```

Заметьте, что все эти подпрограммы вызывают контекстное переключение. Это не строго обязательно для подпрограммы *старт,* но общая разработка упрощается, если все подпрограммы ядра вызывают контекстное переключение.

Необходимо описать еще несколько подпрограмм, относящихся к работе с сигналами и семафорами. Каждая из них требует очереди FIFO, которую в языке RTL/2 можно построить с помощью записей и ссылочных переменных. Структуры

данных для одной очереди можно описать следующим образом:

```
% Очереди задач %
mode ячейка(целый идэадача, ref ячейка следующий);
mode очередь(целый использов, ref ячейка первый, последний);
data списокзадач;
   array(чзадач)ячейка зсписок;
enddata;
```

Вид структуры очереди, которую представляют эти описания, приведен на рис. 11.2. Для каждого сигнала и семафора требуется одна такая очередь. Каждая из них представляется за-

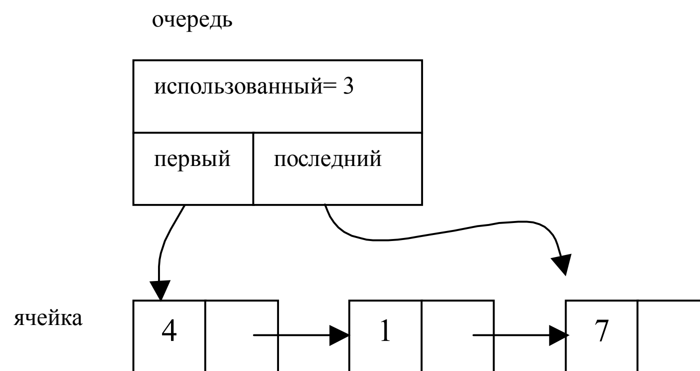


Рис. 11.2. Структура очереди.

писью *очередь,* которая указывает число задач в очереди и содержит указатели на первую и последнюю задачу в очереди. Сама очередь является связанным списком ячеечных записей, где каждая ячейка представляет одну задачу. Так как в каждый момент задача может быть только в одной очереди, требуется не более *чзадач* ячеек. Они описываются как массив в бруске списка задач (заметим, что динамическое распределение данных в RTL/2 не предусматривается).

Для работы с очередью требуются две операции: одна для вывода первой задачи из начала очереди и другая для занесения задачи в конец очереди. Эти операции можно закодировать следующим образом:

```
% Операции для очереди %
proc изочер(ref очередь q) целый;
   целый зид := q.первый.идзадачи;
   q.использов :=q.использов - 1;
   if q.использов > 0 then
      q.первый := q.первый.следующий;
   end;
   return(зид);
endproc;

 proc вочер(ref очередь q, целый идзадачи);
   ref ячейка зячейка := тсписок(идзадачи);
   if q.использов = 0 then
 q.первый := зячейка;
else
      q.последний.следующий := зячейка;
   end;
   q.последний := зячейка;
   q.испольэов : =  q.использов + 1;
 endproc;
```

Отметим, что введя явный счетчик числа ячеек в каждой очереди, мы одновременно решаем проблему инициализации указателей. Другим решением было бы введение пустых ячеек *nil.* В RTL/2 не имеется специальной возможности проверки нулевого значения ссылки.

Используя данные типы очередей, структуры данных для семафоров и сигналов можно построить как массивы элементов, где каждый элемент содержит флаг и очередь.

```
% Структуры сигнала и семафора %
mode элемент = (целый флаг, очередь очер);
                                        % см. разд. 11.4 %.
data сигсем;
   array(чсиг)элемент сиг;
   array(чсем)элемент сем;
enddata;
```

Из определений, приведенных в разд. 6.2, можно увидеть, что *ждать* и *оградить* являются сходными операциями, как и операции *послать* и *освободить.* Поэтому для реализации этих операций можно написать две общие процедуры.

```
proc устэлем(ref элемент e);
  if e.очер.использов =0 then
     e.флаг := 1;
  else
     эстатус(изочер(e.очер)) := активный;
  end;
endproc;

proc восстэлем(ref элемент e);
  if e.флаг = 1 then
      e.флаг :=0
  else
   вочер(e.очер, зработающий);
   зстатус(зработающий) := ожидание;
  end;
endproc;
```

Теперь операции *послать, ждать, оградить* и *освободить* записываются просто.

```
% Операции для сигналов %
ent proc послать(целый s);
   войти( );
   устэлем(сиг(s));
   покинуть( );
endproc;

ent proc ждать(целый s);
   войти( );
   восстэлем(сиг(s));
   покинуть( );
endproc;

% Операции для семафоров %
ent proc оградить(целый s);
   войти( );
   восстэлем(сем(s));
   покинуть( );
endproc;

ent proc освободить(целый s);
   войти( );
   восстэлем(сем(s));
   покинуть( );
endproc;
```

И последнее, сигналы и семафоры должны быть инициализированы пустыми очередями и подходящими флажковыми значениями. У сигналов вначале флаг должен быть сброшен, а у семафоров флаг установлен. Эта инициализация выполняется с помощью следующей процедуры *иницочереди,* которая вызывается процедурой *иницядро*:

```
proc иницочереди( );
  for i := 1 to чсиг do
     ref элемент e :== сиг(i);
        e.флаг :=0; e.очер.использов. := 0;
  rep;
  for i := 1 to чсем do
     rep элемент e :=сем(i);
     e.флаг := 1; e.очер.использов := 0;
  rep;
endproc;
```

Этим и завершается описание ядра супервизора

11.4. ОБСУЖДЕНИЕ

Язык RLT/2 относится к числу лучших языков реального времени, базирующихся на Алголе. Его общее построение относительно просто и надежно, он выгодно отличается от современных ему языков аналогичного вида (например, CORAL 66, PEARL и т. д.). Его наиболее полезным свойством является, безусловно, хорошо определенная структура раздельной компиляции. Кроме того, его дробный тип является одним из наиболее действенных решений проблемы программирования арифметики с фиксированной запятой в языке высокого уровня, встречавшихся до сих пор.

Главная причина, по которой в эту книгу было включено подробное описание RTL/2, состоит в том, что он помогает понять, как развивались современные языки из более традиционных проектов, базирующихся на Алголе. В оставшейся части этого раздела внимание будет обращено на главные недостатки RTL/2; мы хотим наиболее полно осветить те области, в которых недостаточность средств привела к тем изменениям, которые мы встречаем в проектах современных языков.

Во-первых, самым очевидным недостатком RTL/2 является его бедная система типизации. Предоставляется только два дискретных типа, *байтовый* и *целый,* и, так как слабая типизация позволяет использовать эти типы в качестве замены символьных, логических и перечислимых типов, в результате складывается менее чем удовлетворительная ситуация. Проблема слабой типизации была полностью обсуждена в гл. 2, и здесь мы не будем повторяться. Достаточно еще раз обратить внимание на то, что использование примитивного типа, такого, как *целый,* в роли многоцелевого типа ведет к серьезному нарушению надежности.

Даже еще более серьезным недостатком, чем скудность примитивных типов, является в RTL/2 малочисленность средств для описания определенных пользователем типов. Единственным типом, который разрешается определять пользователю, является комбинированный тип. Регулярные переменные должны полностью специфицироваться каждый раз, когда они описываются (описания **let** могут частично спасти положение, но только в простых случаях). Более того, ограничения на регулярные и комбинированные типы, состоящие в том, что компонентами массива не могут быть массивы и компонентами

записи не могут быть записи или массивы записей, являются очень серьезным недостатком.

Недостаточные возможности типизации приводят в конечном счете к тому, что RTL/2 по существу не поддерживает разработку методом абстракции данных. Заметим, что разработка механизмов сигнала и семафора в предыдущем разделе описывалась в терминах абстрактного типа *очередь,* иначе говоря, сигнал и семафор описывались так:

mode *элемент* = (*целый флаг, очередь очер*);

На самом деле в RTL/2 это запрещается. Поэтому в практической реализации структура вида элемент должна быть описана более подробно

mode *элемент* = (*целый флаг, использов,* **ref** *вызов первый, последний*)

где теряется различие между теми компонентами, которые реализуют очередь, и теми, которые представляют состояние. Это достойно сожаления, так как пропадает ценность записи как средства структурирования данных. Впрочем, если быть совсем справедливыми, следует указать, что эти ограничения являются не результатом присущей общему проекту языка слабости, а результатом попытки сделать RTL/2 очень эффективным языком для компиляции и выполнения. То же самое относится и к ограничениям, касающимся локальных переменных и параметров процедур, которые не могут быть структурными типами, и описаний процедур, которые не могут быть вложенными. Тот факт, что все эти ограничения были сняты в проектах современных языков, отражает уменьшение степени важности, которую стали приписывать эффективности, по сравнению с более насущной потребностью иметь надежное хорошо структурированное программное обеспечение.

Вторым аспектом неспособности RTL/2 поддерживать абстракцию данных является недостаток возможностей для управления именной областью действия. Модуляризация полностью ограничивается раздельной компиляцией. Поэтому внутри модуля все объекты брусков данных являются глобально доступными, в результате чего, если нужно ограничить доступ к структурам данных, они должны быть помещены в отдельный модуль. Это становится практически неосуществимым в больших системах программного обеспечения, где это могло бы привести к выделению нескольких сотен логических модулей. К тому же, конечно, хотя имеется возможность экспортировать из модуля процедуры и переменные, нельзя экспортировать типы данных.

Во-вторых, RTL/2 имеет своей целью программирование многозадачных систем, но не содержит специальных мультипрограммных возможностей. Поэтому, хотя и имеется возможность строить многозадачные системы (как было показано в предыдущем разделе), они базируются на примитивных механизмах взаимосвязи, таких, как семафоры и сигналы. В гл. 6 были обсуждены недостатки этого подхода и ясно показаны преимущества использования встроенных конструкций высокого уровня, таких, как мониторы или рандеву. Поскольку в проект RTL/2 не включены специальные возможности для мультипрограммирования, RTL/2 исключает возможность воспользоваться выгодами проверки во время компиляции межзадачного интерфейса и вынуждает обращаться к ненадежным механизмам взаимосвязи низкого уровня.

В-третьих, RTL/2 не поддерживает прямое программирование устройств низкого уровня. Хотя и можно использовать вставки кода для обращения к регистрам устройств, отсутствие и средств для представления процессов, и подходящих типов данных для представления структуры регистров устройств делает невозможным структурный подход к программированию устройств низкого уровня. Из общих принципов языка RTL/2 должно, по-видимому, следовать, что все обращения к регистрам устройств следует осуществлять через операционную систему.

И последнее, RTL/2 обеспечивает мощный механизм обработки ошибок в форме оператора **goto** и меточных и процедурных переменных. Однако этот механизм - очень низкого уровня, его использование в программе при необходимости развернутой обработки ошибок может сильно ухудшить ясность программы. Как было выяснено при обсуждении в гл. 8, для поддержания надежности и удобочитаемости требуется более структурный подход.

Подводя итог, отметим, что RTL/2 обеспечивает вполне приемлемую схему для разработки алгоритмов методом абстракции управления, но недостаточен для поддержания разработки методом абстракции данных. Более того, его возможности для обслуживания многозадачного пропуска (неявные), программирования устройств и обработки ошибок — низкого уровня и, следовательно, ненадежны. В результате, хотя RTL/2 и достаточно гибок, чтобы обеспечить программирование как очень маленьких, так и очень больших систем, в первом случае неприемлемой является его явная зависимость от операционной системы, а во втором случае его недостаточная система типизации и особенности низкого уровня ставят серьезные проблемы надежности и сопровождения.

Все это — главное, что нужно извлечь из уроков, которым учат нас язык RTL/2 и аналогичные ему языки. В проекте RTL/2 есть много других интересных аспектов, достойных обсуждения, например чрезмерное использование ссылочных типов и его числовая система. Но это все же менее существенные аспекты, которых мы уже касались в первой части этой книги. Ключевые недостатки RTL/2 лежат, безусловно, в области абстракции данных и мультипрограммирования. Следующим языком, который мы будем описывать, является Модула; этот язык выражает появление современной методологии проектирования, в которой оба этих недостатка считаются критичными и, следовательно, требующими решения.

Глава 12. МОДУЛА

12.1. ПРЕДПОСЫЛКИ

Язык Модула был разработан Н. Виртом в организации ETH в Цюрихе в 1975 году. Он был создан там же, где и Паскаль, и многие из его главных особенностей взяты из этого языка. Тем не менее язык Модула нельзя считать преемником Паскаля, так как он предназначен для применения в очень специальной области — в области программирования встроенных промышленных и научных вычислительных систем реального времени.

Необычность языка Модула заключается в том, что он проектировался для работы на не оснащенной машине без какой-либо поддержки со стороны операционной системы. У него нет встроенных возможностей ввода-вывода высокого уровня, но он предоставляет конструкции для написания непосредственно на этом языке подпрограмм управления устройствами. Явно сформулированной задачей Модулы был охват тех областей, в которых традиционно доминировало программирование на языке ассемблера. В этом отношении язык Модула особенно удобен для небольших систем, ориентированных на микропроцессоры; он получил распространение в управлении процессами, обработке массивов данных, интеллектуальном выводе и управлении потоком сообщений.

С Модулой не связывается определенный формальный стандарт, опубликованные Виртом первоначальные спецификации использовались реализаторами и пользователями языка в качестве стандарта de facto (Вирт, 1977a). Первая цюрихская реализация Модулы была выполнена для машины PDP 11

(Вирт, 1977с), появились и другие реализации для этой машины (например, Коттам, 1980; Смедена и др., 1979). В какой-то мере проект Модулы оказался под влиянием архитектуры PDP 11, но были реализации и для других машин (например, Нолан и др., 1979). При описании Модулы в данной главе предполагается, что целевой машиной является PDP И.

12.2. ЯЗЫК

Предоставляемые Модулой возможности можно разделить на две группы. Во-первых, предоставляются возможности последовательного программирования, аналогичные возможностям стандартного Паскаля. При этом некоторые из особенностей Паскаля были опущены, а основным новшеством в Модуле явилась конструкция модуля для абстракции данных. Во-вторых, предоставляются конструкции для параллельного программирования в форме процессов и мониторов. В следующих разделах дается краткий обзор языка. Сначала описываются особенности последовательного программирования, а затем — параллельного.

12.2.1. Особенности последовательного программирования

Для языка Модула характерна система сильной типизации, базирующаяся на структурной эквивалентности типа. Имеются три простых дискретных типа: *целый, логический* и *символьный.* Кроме этого, с помощью перечисления можно ввести определенные пользователем дискретные типы. В Модуле нет никакого механизма для описания поддиапазонов дискретных типов.

Структуризация данных в Модуле осуществляется с помощью массивов и записей. Массивы могут быть любой размерности, индексы — любого дискретного типа; нет никаких ограничений на тип компонент. Динамические массивы не поддерживаются. Записи специфицируются с помощью обычных обозначений Паскаля. Нет никаких ограничений на типы компонент записи, но вариантные части записей не поддерживаются.

В добавление к этому в Модуле есть предопределенный тип, называемый *биты,* который обозначает массив логических значений фиксированной длины. Предполагается, что этот массив упаковывается в одно машинное слово, т. е.

type *биты* = **array** 0 : *w* **of** *логический*;

где *w* — размер слова целевой машины в битах минус единица. Литеральные значения типа *биты* обозначаются списком тех индексов, для которых компоненты истинны. Например, [[O, 3:7, 15] обозначает значение типа *биты,* у которого биты

0, 3, 4, 5, 6, 7 и 15 истинны (т. е. установлены), а все остальные биты ложны. Заметьте, что 3:7 является сокращенной формой записи 3, 4, 5, 6, 7.

Описания объектов соответствуют формату Паскаля. Разрешаются описания констант, типов и переменных и в отличие от Паскаля они могут приводиться в произвольном порядке. В качестве примера описания объектов в Модуле рассмотрим следующий фрагмент:

```
(* Терминальные объекты *)
const lf = 12c; cr=15c; звездочка = '*';
        ширинастроки = 80;
type строка = array 1 : ширинастроки of символ];
var cм: символьный;  команда; строка;
(* Объекты вывода *)
type видцвета = (красный, желтый, зеленый, синий);
      точка = record
                    определенный: логический;
                    цвет: видцвета;
                    интенсивность: целый
                 end;
var изображение : array 0 : 63, 0 : 63 of точка;
     элемент: точка;
```

Прежде всего заметим, что ключевые слова **type** и **var** встречаются более одного раза. Модула разрешает включать в описательной части любое число разделов **const, var и type,** Единственное ограничение состоит в том, что все идентификаторы должны быть определены перед тем, как они используются. Константы *lf* и *cr* имеют шестнадцатеричные значения 12 и 15 соответственно; они типа *символьный.* Эти обозначения используются для определения символов, которые не выдаются на печать, более обычная форма символьной константы показана в описании *звездочка.* Тип *строка* является одномерным массивом символов, индексы которого пробегают значения от 1 до *ширинастроки.* Доступ к компонентам типа строка осуществляется с помощью обычной индексной нотации, например *команда* [1] обозначает первую компоненту регулярной переменной *команда.* Тип *видцвета* является перечислительным типом с четырьмя значениями: *красный, желтый, зеленый* и *синий.* Тип *точка* — запись с тремя компонентами. Для обозначения компонент записи, как обычно, используется точка, например, *элемент.цвет* обозначает компоненту *цвет* комбинированной переменной *элемент.* Кроме этого, компоненты записи могут

быть указаны непосредственно раскрытием записи с помощью оператора **with,** например,

```
with элемент do
  определенный := истина;
  цвет := красный
end
```

И наконец, переменная *изображение* является двумерным массивом *точек.*

В Модуле имеются четыре класса операций с различными приоритетными уровнями. На самом нижнем уровне находятся операции отношения <, >, < =, >=, < >. Они могут применяться к *целым, символьным, логическим* и *перечислительным* типам, возвращая логический результат. Кроме того, к типам *биты* могут применяться операции = и <>. На следующем уровне — аддитивные операции +, —, **or** и **xor**. За ними следуют операции класса умножения *, /, **div**, **mod** и **and**. И на самом высшем уровне находится операция отрицания **not.** Арифметические операции +, —, /, *, **div** и **mod** применяются к целым операндам и возвращают целый результат. Операция / обозначает деление с отбрасыванием дробной части. Операции **div** и **mod** возвращают частное $q = x$ **div** y и остаток $r = x$ **mod** y, такие, что $x = q*y + r$, $0 \le r < y$. Делитель y должен быть положительным. Логические операции **or, xor, and** и **not** имеют обычный смысл и применяются к логическим операндам, возвращая логический результат. Кроме того, они могут быть применены к типу *биты*; *в* этом случае специфицируемая операция применяется ко всем соответствующим элементам операндов, например

[0, 1, 2] **or** [0, 6] = [0, 1, 2, 6]

В Модуле предусмотрены пять видов операторов управления, все они имеют явные закрывающие скобки. Операторы **if, case, while и repeat** традиционны. Их форма иллюстрируется следующими примерами:

```
(a) if x < O then
        S1
    elsif x = 0 then
        S2
    else
        S3
    end

(b) case см of
    'A' : begin S1 end;
    'B' : begin S2 end;
    'C' : begin S3 end
    end;

(c) while b do
        S1
    end
```

(d) **repeat**
*S*1
until *b*

где *Si* обозначает последовательность операторов, разделенных точками с запятой. В операторе **if** части **elsif** и **else** не обязательны, может быть включено любое число частей **elsif**.

Пятым оператором управления в Модуле является конструкция **loop**, которая имеет форму;

loop
*S*1 **when** *B*1 **do** *X*1 **exit**
*S*2 **when** *B*2 **do** *X*2 **exit**
.
.
.
Sn **when** *Bn* **do** *Xn* **exit**
S
end

где *Si* и *Xi* обозначают последовательности операторов, а *Bi* обозначают условия выхода. Когда вычисленное значение *Bi* есть *истина,* выполняется последовательность *Xi* и затем цикл завершается. Часть **do** может быть опущена, если никаких завершающих действий выполнять не требуется. В Модуле нет оператора **for**.

Процедуры в Модуле аналогичны процедурам Паскаля. Параметры передаются либо по значению, либо по ссылке, последнее указывается с помощью ключевого слова **var**. Для обозначения функции нет специального ключевого слова, процедура может возвращать результат. Описания процедур могут быть вложенными, применяются обычные правила открытой области действия. Впрочем, процедура может содержать список **use** в своем заголовке; этот список специфицирует те описанные во внешних блоках идентификаторы, которые используются в этой процедуре. При включении списка **use** никакие другие внешние идентификаторы, кроме перечисленных в списке, не доступны. Следующие два примера иллюстрируют формат процедур в Модуле:

(a) **procedure** *писатьстроку*(**var** *l*: *строка*);
use *ширинастроки, писатьсм*;
var *i*: *целый*;
begin
i := 1;
while *i* < = *ширинастроки* **do**
писатьсм(*l*[*i*]);
приращ(*i*)
end
end *писатьстроку*;

```
(b) procedure максцел(x, y : целый): целый;
        use;
        begin
          if x > y then
            максцел := x
          else
            максцел := y
          end
        end максцел;
```

В примере (а) процедура *писатьстроку* использует два внешних имени: константу *ширинастроки* и процедуру *писатьсм.* Внутри процедуры *писатьстроку* никакие другие внешние имена не видимы. Процедура *прираш*(i) является стандартной процедурой Модулы и означает $i := i + 1$; она доступна во всей программе. В примере (b) *максцел* является функцией, которая возвращает значение. Пустой список **use** означает, что никакие другие внешние имена не видимы; поэтому *максцел* является чистой процедурой. Процедура *писатьстроку* вызывается оператором обращения к процедуре, например

писатьстроку(*моястрока*)

в то время как процедура *максцел* вызывается как функция

z := *максцел*(i, 1000)

Отметим также, что имя процедуры должно задаваться как в конце описания процедуры, так и в его начале. Это требуется для того, чтобы увеличить удобочитаемость и чтобы компилятор смог выдать более точную диагностику при отсутствии закрывающей скобки **end**.

Теперь осталось описать еще одну языковую последовательную особенность — модуль. За исключением одного важного упрощения, включенный в Модулу механизм модуля идентичен механизму, описанному в гл. 5. Упрощение состоит в том, как специфицируется интерфейс модуля. В Модуле все объекты, которые необходимо импортировать в модуль, должны быть перечислены в списке использования, а все объекты, которые

необходимо экспортировать из модуля, должны быть перечислены в списке определения. Впрочем, приводятся только имена транспортируемых объектов. Чтобы удостовериться, какие виды объектов обозначают эти имена (например, константы, типы, процедуры и т. д.), необходимо просмотреть основное тело модуля. Поэтому в Модуле не представляется возможным отделить спецификации интерфейса от реализационной части. Любой объект может импортироваться или экспортироваться через границу модуля, но если переменная экспортируется, то вне модуля ее можно только читать.

В качестве примера модуля языка Модула рассмотрим модуль стек, приведенный в разд. 5.3.1; он кодируется следующим образом:

```
module стек;
   use элемент, (* тип заносимых в стек элементов *)
   define втолкнуть, вытолкнуть; (* стековые операции *)
   var s : array 1 : 100 of элемент;
        sp : целый;
   procedure втолкнуть(x: элемент);
   begin
     sp := sp + 1; s[sp] := x
   end втолкнуть;
   procedure вытолкнуть(var x : элемент);
   begin
     x := s[sp]; sp := sp— 1
   end вытолкнуть;
begin
   sp := 0 {* инициализация *)
end стек;
```

Модуль является в языке Модула первичной программной конструкцией. Внешний модуль определяет главную программу. Поэтому общая форма программы в Модуле имеет вид:

```
module имяглавнойпрограммы;
   (* описания констант, типов, переменных, процедур и
                                                  модулей *)
begin
   (* код главной программы *)
end имяглавнойпрограммы
```

Описания процессов, которые здесь опущены, рассматриваются в следующем разделе.

12.2.2. Особенности параллельной обработки

Процессы могут быть описаны в Модуле с помощью нотации, аналогичной нотации для процедур. Например

```
process передать(канал : целый);
   use получить, выдать;
   var c : сим;
begin
   loop  получить(канал, c); выдать(канал, c);
   end
end передать;
```

определяет процесс, который выполняет бесконечный цикл, получая и выдавая символы через специфицируемый канал. Экземпляры процессов создаются операторами обращения к процессам, которые синтаксически идентичны обращениям к процедурам, например,

передать(1); *передать*(2);

вызвало бы инициацию двух экземпляров процесса *передать,* первый из них передавал бы поток символов через канал 1 и второй — через канал 2. Процессы разрешается описывать только на самом внешнем уровне программы, т. е. они не могут быть вложенными в описания процедур. Соответственно, операторы вызова процессов могут встречаться только на самом внешнем уровне и, следовательно, ограничены телами модулей. Имеющийся в Модуле механизм межпроцессной связи базируется на понятии монитора. Если описанию модуля предшествует ключевое слово **interface,** то модуль получает свойства монитора, т. е. внешние обращения к процедурам внутри модуля удовлетворяют правилу взаимного исключения. Процессы могут синхронизироваться с помощью сигналов. Для сигналов определены три операции:

(i) *ждать*(*s*) — вызывающий процесс задерживается до тех пор, пока не будет получен сигнал.
(ii) *послать*(*s*) — сигнал *s* посылается первому процессу, ждущему в очереди, связанной с *s*.
(iii) *ждет*(*s*) — логическая функция, которая возвращает значение *истина,* если есть по крайней мере один процесс, ждущий *s*.

Выполняющие оператор *ждать* процессы, как правило, обслуживаются в порядке FIFO. Впрочем, им могут быть присвоены приоритеты, специфицирующие ранг оператора *ждать,*

например, *ждать*(*s,n*) вызовет задержку выполняющегося процесса с рангом *n*. Следующие процессы, которые выполняют операцию ожидания с рангом *m*, получают сигнал *s* прежде процесса с рангом *n* тогда и только тогда, когда $m < n$. Когда ранг не специфицирован, предполагается ранг 1 (т. е. самый высокий приоритет).

Необычным свойством модульного интерфейса является то, что выполнение операций *ждать* и *послать* не подчиняется правилу взаимного исключения. Поэтому если процесс выполняет оператор *ждать* в интерфейсной процедуре, то другой процесс может войти в интерфейсный модуль[10], если даже ждущий процесс все еще находится внутри модуля. Кроме этого, когда процесс выполняет оператор *послать* в интерфейсном модуле, получающему процессу гарантируется взаимно исключающий доступ к этому модулю, а посылающий процесс задерживается до тех пор, пока получающий процесс не завершит свою интерфейсную процедуру. Эта семантика отличается от семантики традиционного монитора, так как обычно посылающий процесс продолжает завершение своей интерфейсной процедуры, прежде чем получающему процессу разрешается возобновить свою работу. Семантика Модулы является необходимым следствием стратегии реализации, используемой на однопроцессорных машинах. К этому мы вернемся позднее в разд. 12.4, а вытекающие из этого следствия обсуждаются в разд. 12.5.

В качестве примера рассмотрим следующий интерфейсный модуль, который может быть использован для реализации логического семафора:

```
interface module семафор;
  define оградить, освободить;
  var занят : логический;  свободен : сигнал;
  procedure оградить;
  begin
    if занят then ждать(свободен) end;
    занят :=истина
 end оградить;
  procedure освободить;
  begin
   занят := ложь; послать(свободен)
end освободить;
 begin
   занят := ложь
 end семафор;
```

[10] Т.е. модуль с описателем **interface.** — *Прим. ред.*

Обычно процессы могут выполнять процедуры *оградить* и *освободить* только в режиме взаимного исключения, поэтому целостность логической переменной *занят* гарантирована. Однако если процесс выполняет операцию *ждать* в процедуре *оградить,* то взаимное исключение отменяется и разрешается вход другому процессу. Когда процесс выполняет операцию *послать,* возобновляется выполнение первого ждущего процесса и правило взаимно исключенного доступа к модулю начинает работать снова.

Наряду с интерфейсными модулями Модула обеспечивает также модули устройств, которые предоставляют непосредственный доступ к машинному оборудованию. Модуль устройства обладает такими же мониторoподобными свойствами, как и интерфейсный модуль, но кроме этого содержит зависящие от машины процессы устройств. В качестве примера рассмотрим следующий модуль устройства, который реализует простые часы реального времени на PDP 11. При каждом прерывании часов, происходящем с постоянной частотой, модуль посылает сигнал *такт* всем ждущим процессам (ср. пример модуля часов из разд. 7.3.3).

```
device module чрв[6];
  define такт;
  var такт: сигнал;
  process часы[100B];
     var регстатус[177546B]: биты;
  begin
     регстатус[6] := истина;
     loop
        выпев;
        while ждет(такт) do
           послать(такт)
        end
      end
   end часы;
 begin
   часы
 end чрв;
```

В заголовке модуля устройства специфицируется приоритет прерывания, с которым должны выполняться все команды внутри модуля. В теле модуля определяются сигнал *такт* и процесс устройства. Значение 100В (В означает восьмеричное число) указывает адрес вектора прерывания, с которым связан процесс устройства. Тело процесса устройства состоит из бесконечного цикла:

loop *выпвв*; *послать такты* **end**

Оператор *выпвв* является предопределенной операцией в Модуле. Выполнение оператора *выпев* приводит к задержке вызова процесса устройства аналогично выполнению операции *ждать* для сигнала. Впоследствии он возобновляется при возникновении соответствующего прерывания машинного оборудования. Поэтому в приведенном выше примере данный цикл будет выполняться один раз для каждого появления прерывания от часов с равномерной частотой.

Следует заметить, что посылка сигналов процессом устройства не подчиняется правилам для посылки сигналов обычными процессами внутри интерфейсных модулей. Когда процесс устройства посылает сигнал, он не нарушает правила взаимно исключающего доступа к модулю, но продолжает работу до тех пор, пока не будет выполнен оператор *ждать* или *выпев.* Следовательно, процессы устройств подчиняются традиционной семантике монитора.

Доступ к регистрам устройств машинного оборудования осуществляется с помощью явного указания отображения переменных на физическую память. Например, описание

var *регстатуса*[177546B] : *биты*;

вводит переменную с именем *регстатуса* типа *биты,* которая должна быть помещена по восьмеричному адресу 177546. Это есть ячейка регистра статуса аппаратных часов на PDP 11. Оператор

регстатуса[6] := *истина*

таким образом, установит в результате шестой бит этого регистра статуса, который является битом обеспечения прерывания. Отображения такого вида можно специфицировать только внутри модулей устройств.

На использование процессов устройств Модула накладывает четыре главных ограничения:

(i) Процессы устройств не могут посылать сигналы другим процессам устройств.
(ii) Процессы устройств не могут обращаться к нелокальным процедурам,
(iii) Можно активировать только один экземпляр процесса устройства.
(iv) Операторы *ждать* в процессах устройств могут быть только ранга 1, т. е. самого высокого приоритета.

Эти ограничения и ограничения для обычных процессов, про которые говорилось ранее, введены для того, чтобы сделать возможной простую и эффективную реализацию программ Мо-

дулы. Однако, как будет показано позднее, такая организация процессов в Модуле ведет к ряду неприятных ограничений языка.

12.3. ИСПОЛЬЗОВАНИЕ МОДУЛЫ

12.3.1. Составление очереди задач в реальном времени

Чтобы проиллюстрировать, как используется Модула в типовых ситуациях, опишем обобщенную схему простой программы составления очереди задач в реальном времени. Эта система аналогична системе, описанной Холденом и Вандом (1977),

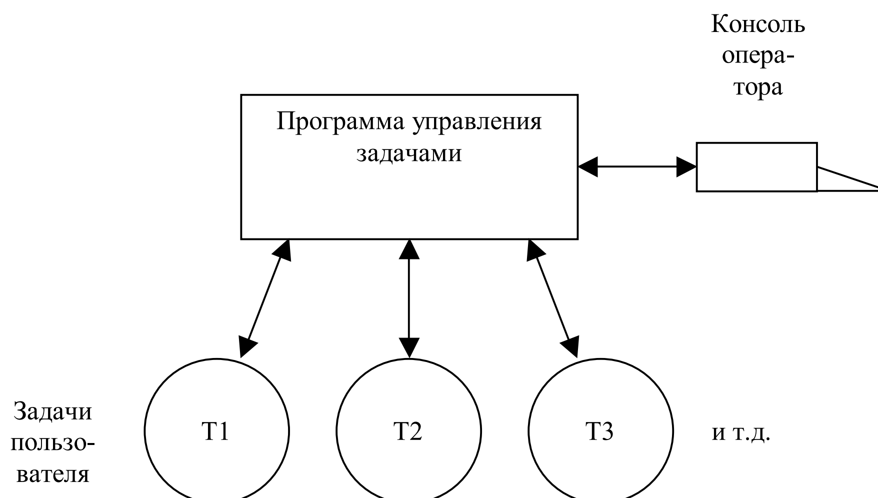


Рис. 12.1. Конфигурация многозадачной системы.

которая сама базировалась на системе Бринча Хансена (1975b). Система Хансена была написана на Параллельном Паскале.

Структура этой системы приведена на рис. 12.1. Она состоит из набора пользовательских задач, которые управляются модулем управления задачами в ответ на команды, поступающие с консоли оператора. Каждая задача идентифицируется целым числом в диапазоне от 1 до *максзадач.* Имеются три операторские команды, каждая из которых идентифицируется одной буквой:

(a) T t Установить системные часы на время *t*
(b) S n t Начать выполнение задачи *n* с периодом *t*
(c) Q n Остановить задачу *n*

Наряду с предоставлением средства ввода для операторских команд консоль оператора может быть также использована задачами пользователей для чтения и выдачи. Поэтому ввод от оператора программе управления задачами сопровождается нажатием сигнальной клавиши на клавиатуре консоли. Программа управления задачами отвечает на это выдачей указания на дальнейший ввод. Так, например, может быть использован

следующий диалог для запуска задачи 1 и задачи 3 с периодами 5 сек и 1 час соответственно. Вначале устанавливается время на 9 час 30 мин.

Ввести команду: *T* 9 : 30 : 0
Команда принята
Ввести команду: *S*10 : 0 : 5
Команда принята
Ввести команду: *S*30 : 1 : 0
Команда принята

Вслед за этими командами будет происходить непрерывное выполнение задач 1 и 3 со специфицированными периодами,. Каждая задача записана как процесс Модулы в форме

```
process задачапользов;
   const идзадачи = n;
begin
   включит ь(идзадачи);
   loop
      приостановить(идзадачи);
        …
      (* действия, выполняемые задачей *)
        …
   end
end задачапользов;
```

Обращение к *включить* сообщает программе управления задачами о факте существования задачи. Обращение к *приостановить* вызывает приостановление задачи до тех пор, пока не пройдет специфицируемый для нее период.

В практической работе задачам пользователя может потребоваться доступ к различным видам устройств ввода-вывода и необходимость связи друг с другом. Однако в нашем примере каждая задача может рассматриваться как работающая самостоятельно и выполняющая операции ввода-вывода только на консоли оператора. На самом деле все, что делает каждая задача, - это распечатка своего идентификатора и текущего времени на консоли оператора каждый раз, когда она начинает выполняться.

В остальной части этого раздела описывается, как на самом деле строится в Модуле программа управления задачами. Как и следует ожидать, она состоит из одного модуля, внутри которого имеется набор модулей и процессов, реализующих все необходимые функции. Чтобы проиллюстрировать методологию разработки, которую поддерживает Модула, модуль программы управления задачами будет описываться так, как если бы мы начинали разрабатывать с самого начала,

Прежде всего можно выделить четыре различные функции, которые должна обеспечить программа управления задачами:

(i) составление расписания для задач пользователя
(ii) отслеживание времени
(iii) терминальный ввод-вывод
(iv) диалог с оператором

В следующих четырех подразделах рассматривается по очереди каждая из этих функций.

12.3.2. Составление расписания для задач пользователя

Составление расписания для задач пользователя требует, чтобы в центральной таблице хранилась определенная информация. Каждая запись в таблице должна регистрировать состояние задачи как активное или неактивное; время, когда ее нужно запускать в следующий раз; интервал между последовательными выполнениями задачи; сигнал для постановки в очередь, когда задача неактивна. Кроме того, следует ожидать, что не все записи таблицы будут использоваться в конкретном применении. Поэтому должно быть записано еще одно состояние, обозначающее, существует или не существует задача на самом деле. Это состояние будет называться *общий.* Подходящей структурой данных для хранения этой информации будет массив записей, где каждая запись имеет следующий вид:

```
type данныезадачи = record
                        общий, активн : логический;
                        когда, интервал: время;
                        очередь: сигнал
                    end
```

Заметим, что на этом этапе нам не следует интересоваться тем, как будет время представляться фактически. Поэтому вводится абстрактный тип данных с именем *время.* Операции, которые нужны для работы с объектами типа *время,* будут введены по мере надобности.

Доступ к информации *данные задачи* нужен для трех различных системных компонент: задач пользователя, операторского интерфейса и фактической подпрограммы составления расписания. Так как каждая из этих компонент может выполняться параллельно, данные задачи должны быть защищены внутри интерфейсного модуля и доступ к ним следует ограничить множеством интерфейсных процедур. Теперь можно привести полный текст этого интерфейсного модуля:

```
Interface Module таблицазадач;
   define старт, стоп, пересоставить, включить, приостановить;
   use время, выбвремя, добвремя, нераньше;
   const максзадач = 16;
   type данныезадачи = record
                         общий, активн : логический;
                         когда, интервал : время;
                         очередь: сигнал
                       end;
  var статусзадачи : array 1 : максзадач of данныезадачи;

  procedure корзадача(идзадачи : целый; var ок : логический);
  begin
     ок := (идзадачи > 0) and (идзадачи < = максзадач)
  end корзадача;

  procedure старт(идзадачи : целый; период : время;
                                    var ок: логический);
  begin
     корзадача(идзадачи, ок);
     if ок then
        with статусзадачи[идзадачи] do
           if общий then
              активный := истина; интервал := период;
              выбвремя(когда)
           else
            ок:= ложь
           end
        end
     end
  end старт;

  procedure стоп(идзадачи : целый; var ок: логический);
  begin
     корзадача(идзадачи, ок);
      if ок then
        статусзадачи[идзадачи].активн := ложь
      end
  end стоп;

  procedure пересоставить;
    var идзадачи ; целый;
         теквремя: время;
  begin
     идзадачи :=1; выбвремя(теквремя);
     repeat
        with статусзадачи[идзадачи] do
          if активн and нераньше(когда, теквремя) then
             добвремя(когда, интервал);
            послать(очередь)
          end
        end; приращ(идзадачи)
      until идзадачи > максзадач
    end пересоставить;
```

```
procedure включить(идзадачи : целый);
begin
   статусзадачи[идзадачи].общий := истина
end включить;

procedure приостановить(идзадачи : целый);
begin
   ждать(статусзадачи[идзадачи].очередь);
end приостановить;

procedure иниц;
   var идзадачи ; целый;
begin
  идзадачи := 1;
  repeat
     with статусзадачи[ид задачи} do
    общий := ложь; активн := ложь
     end
     приращ(идзадачи)
   until идзадачи > максзадач
end  иниц;

begin
  иниц
end таблицазадач;
```

Интерфейс с оператором обеспечивают процедуры *старт* и стоя. *Старт* задает переменной *активн* значение истина, переменной *интервал* - значение, равное специфицированному периоду, и переменной *когда* — значение текущего системного времени, которое определятся процедурой *выбвремя,* описываемой позднее. Заметим, что процедура *старт* перед фактическим запуском задачи проверяет как корректность задачи (т. е. номер задачи), так и то, что она находится в состоянии *общий.* Процедура *стоп* просто задает переменной *активн* значение ложь.

Процедура *пересоставить* вызывается программой составления расписания регулярно через определенные интервалы времени и проверяет статус каждой задачи. Если у задачи значения переменных *активн* и *когда* истинны, то она возобновляется посылкой ей сигнала *очередь.* Заметим, что в процедуре *пересоставить* предполагается существование двух процедур, обрабатывающих объекты типа *время* : *нераньше* — для сравнения значений времени *доввремя* — для увеличения значения переменной *когда* на значения интервала.

Процедуры *включить* и *приостановить* вызываются самими задачами пользователя так, как описано в разд. 12.3.1. *Включить* задает переменной *общий* значение истина, а *приостановить*

выполняет операцию *ждать* от имени вызывающей задачи.

И последнее, фактическое составление расписания осуществляется процессом, который выполняется один раз в секунду,. Он кодируется совсем просто:

```
process прогсосрас
   use такт, пересоставить
begin
   loop
      ждать(такт); пересоставить
   end
end прогсосрас;
```

Сигнал *такт* выдается следящей за временем компонентой один раз в секунду.

12.3.3. Слежение за временем

Следящая за временем компонента системы выполняет две функции. Во-первых, она обеспечивает поддержание текущего» времени суток, во-вторых, она выдает сигнал *такт* каждую секунду. Реализация первой функции относительно проста, ее код. можно привести сразу же:

```
interface module счетемчасы;
   define выбвремя, установвремя;
   use время, иницвремя, добвремя, такт;
   var системвремя, однасек : время;

   procedure установвремя(t : время);
   begin
      системвремя := t;
   end установвремя;

   procedure выбвремя(var t: время);
   begin
      t := системвремя
   end выбвремя;

   process часы;
   begin
      loop  ждать(такт); добвремя(системвремя, однасек)
      end
   end часы;

 begin
   иницвремя(системвремя, 0, 0, 0);
   иницвремя(однасек, 0, 0, 1);
   часы
end системчасы;
```

Время суток хранится в переменной *системвремя,* над которой производят операции с помощью процедур *установвремя* и *выбвремя,* обеспечивающих запись и считывание значения этой переменной, и процесса *часы,* который каждую секунду увеличивает ее значение (ср. процесс составления расписания). Заметим, что инициализация временных переменных должна производиться операциями, экспортируемыми модулем, который реализует абстрактный тип данных *время.*

Интерфейс с аппаратными часами требует введения модуля устройства. Программа управления часами для аппаратных часов PDP 11 была уже описана в разд. 12.2. Точно такая же схема используется и здесь с тем исключением, что сигнал *такт* выдается только один раз в секунду (т. е. один раз на каждые 50 прерываний).

```
device module аппчасы[6];
  define такт;
  var такт: сигнал;

  process прогуправ[100B];
    var счетчик : целый; регстатус[177546B] : биты;
    begin
      счетчик := 0; регстатус[6] := истина;
      loop
        выпев;
        счетчик := (счетчик + 1) mod 50;
        if счетчик = 0 then
          while ждет(такт) do послать(такт)  end
        end
      end
    end прогуправ;

 begin
  прогуправ
 end аппчасы;
```

Здесь следует заметить, что процесс *часы* в модуле *системчасы* логически избыточен (что относится также и к приведенной ранее программе составления расписания). Теоретически процесс устройства мог бы наращивать системное время непосредственно, экономя на контекстном переключении для запуска процесса часов. Однако, к сожалению, Модула не разрешает процессам устройств обращаться к нелокальным процедурам. Поэтому, если даже в модуль устройства *аппчасы* была включена бы переменная *системчасы,* она по-прежнему не могла бы наращиваться процессом устройства, так как представлена абстрактным типом данных, который доступен только через обращения к своим собственным (нелокальным) процедурам.

Теперь, когда сам механизм обработки времени построен, становится удобным определить абстрактный тип *время.* Имеются два достаточно очевидных представления, которые можно было бы использовать. Во-первых, время может быть представлено в виде одного целого, обозначающего время суток в секундах; во-вторых, оно может быть представлено в виде записи, вкоторой явно указаны часы, минуты и секунды, т. е. Либо

```
type время = целый
```

либо

```
type время = record
                 часы, мин, сек : целый
             end
```

Как правило, первый выбор предпочтительнее, так как, хотя и потребуются преобразования во внешнюю форму «часы : мин : сек», все остальные операции, такие, как *сложение* и *сравнение,* выполняются совсем просто. Но, к сожалению, рассчитанные на двадцать четыре часа часы не могут быть представлены в секундах 16-битовым целым числом. Следовательно, должна быть выбрана вторая альтернатива.

Модуль *типвремя* кодируется следующим образом. Отметим, что процедуры чтения и записи определены для объектов типа *время* наряду с остальными операциями, которые использовались ранее. Эти операции обращаются к процедурам чтения и записи, которые описываются в следующем разделе.

```
module типвремя;
  define время, добвремя, нераньше, инщвремя, читатьвремя,
                                                  писатьвремя;
  use писатьцел, читатьцел, писатьсим, читатьсим;
  type время = record
                  часы, мин, сек : целый
               end;

  procedure добвремя(var t: время; i: время);
    var ски, мки, чки: целый;

    procedure модсум(var x, сииз; целый; y, сив,
                                              модуль: целый);
      var сум: целый;  begin
       сум := x + y + сив;
       x := сум mod модуль; сииз := сум div модуль
    end модсум;
   begin
    модсум(t.сек,ски, i.сек,0,60);
    модсум(t.мин, мки, i.мин, ски, 60);
    модсуму(t.часы, чки, i.часы, мки, 24)
   end добвремя;
```

```
   procedure нераньше(t1, t2 : время) : логический,
begin
   нераньше := ложь;
   if t1.часы < t2. часы then
     нераньше := истина
   elsif t1 .часы = t2.часы then
      if t1.мин < t2.мин then
        не раньше := истина
     elsif t1.мин = t2.мин then
        не раньше := t1.сек < = t2.сек
      end
    end
end нераньше;

procedure иницвремя(var t : время; ч,м,с : целый);
begin
  with t do
      часы := ч; мин := лг; сек := с
   end
end иницвремя;

procedure писатьвремя(t : время);
begin
   with t do
     писатьцел(часы); писатьсим(‘:’);
     писатьцел(мин);   писатьсим(‘:’);
     писатьцел(сек)
   end
end писатьвремя;

procedure читатьвремя(var t : время; var ок : логический);
    var с : сим;
begin
  with t do
      читатьинт(часы, ок);
      if ок then
        читатьсим(с);
          if с = ’:’ then
              читатьинт(мин, ок);
              if ок then
              читатьсим(с);
                 if с = ’:’ then
                     читатьцел(сек, ок);
                     if ок then
                         ок := (часы<24) and (мин <60) and (сек < 60)
                     end
                   else
                    ок:= ложь
                   end
                end
              else
                ок:= ложь
              end
            end
          end
   end читатьвремя;
end типвремя;
```

На примере модуля *типвремя* очень хорошо видны все преимущества разработки системы с помощью абстрактных типов данных. Во-первых, все компоненты, использующие *время,* разрабатывались без какого-либо учета мелких подробностей работы с объектами типа *время.* Во-вторых, ни одна из этих компонент не может испортить объект этого типа (например, занося в поле часы значение 99), так как внутренние поля его недоступны. В-третьих, так как все относящиеся к типу *время* подробности внутреннего представления спрятаны в одном модуле, его можно легко заменить. Следовательно, при переносе программы на 32-битовую машину данное базовое представление можно легко заменить на представление в форме одного целой) числа с гарантией, что такое изменение не окажет влияния на функционирование других компонент системы.

12.3.4. Терминальный ввод-вывод

В Модуле нет встроенного ввода-вывод а, поэтому терминальный ввод-вывод должен быть запрограммирован вплоть до уровня аппаратуры консоли PDP 11. Разработку терминального ввода-вывода лучше всего выполнять на двух уровнях. Во-первых, можно предоставить множество процедур высокого уровня для выдачи и чтения символов, строк и целых чисел, используя два примитива низкого уровня *занестисим л выбратьсим* для передачи символов на экран консоли и их выбора с клавиатуры консоли соответственно. Эскизы этих процедур мы приведем без особых комментариев, заметим только, что для операций чтения нужно смотреть на один символ вперед. Эти процедуры могут выполняться параллельно, они заключены в интерфейсный модуль. Кроме этого, используются два семафора *оградить* и *освободить,* обеспечивающие процессу исключительный доступ к консоли.

```
interface module терминальный;
  define запрос, освободить, писатьсим, читатьсим,
         читатьцел, писатьцел, писатстр, кнцстр;
  use выбратьсим, занестисим;
  const lf=12c; cr=15c;
  var свободный : сигнал;
     занятый: логический;
     следсим : сим;

  procedure запрос;
  begin
   if занятый then ждать(свободный) end; занятый := истина
  end запрос;

  procedure освободить;
  begin
   занятый := ложь; послать(свободный)
  end освободить;

  procedure писатьсим(c : сим);
  begin
   занестисим(c);
   if c = cr then занестисим(lf) end
  end писатьсим;

 procedure читатьсим(var c : сим);
 begin
   c := следсим; выбрать(следсим); писатьсим(следсим)
 end читатьсим;

  procedure писатьцел(i: целый);
    var инпосзн : array 1 : 5 of сим;
      n, знач: целый;
 begin
   знач := 1; n = 0;
    repeat
      приращ(n); инпосзн[n] := сим(знач mod 10 + целый('0'));. знач
      :== знач/10
   until знач = 0;
   repeat
     занести(инпосзн[n]); дес(n)
   until n = 0;
 end писатьцел;

 procedure писатьстр(строка : array целый of сим);
   var t: целый;
 begin
  i := низкий(строка);
   while i < = высокий(строка) do
     писатьсим(строка[i]); приращ(i)
   end
 end писатьстр;
```

```
  procedure читатьцел(var i: целый; var ок: логический);
    var c: сим;
  begin
    repeat читатьсим(c) until c < > ‘ ’;
    ок := (c > = ‘0’) and (c < = ‘9’);
    if ок then
      i := 0;
      loop
  i = i * 10 + целый(c) - целый(‘0’);
  when(следсим < ‘0’) or (следсим > ‘9’) exit;
  читатьсим(c)
       end
     end
   end читатьцел;

   procedure кнцстр : логический;
   begin
     кнцстр := следсим = cr
  end кнцстр;

begin
   занятый := ложь; следсим := cr
end терминальный;
```

Процедуры низкого уровня *занестисим* и *выбратьсим* реализуются с помощью традиционной схемы буферизированной передачи одиночного символа, как было объяснено в разд. 7.4.1. Впрочем, заметим, что интерфейс клавиатурой также передает сигнал с именем *звонок* всегда при нажатии клавиши звонка. Он используется для вызова операторского диалога (см. следующий раздел).

```
device module экран[4];
  define занестисим;
  const размербуф = 128;
  type буфер = array 1 : размербуф of сим;
  var хв, хиз, использов : целый;
       буф: буфер;
       неполный, непустой : сигнал;

    procedure занестисим(c : сим);
    begin
      if использов = размербуф then ждать(неполный) end;
      буф[хв] := c; хз := хв mod размербуф + 1;
      приращ(использов); послать(непустой)
     end занестисим;
```

```
    process упрэкран[64B];
      var регстатуса[ 177564B] : биты;
          регданных[177566B] : сим;
     begin
       loop
         if использов = 0 then ждать(непустой) end;
         регданных :== буф[хиз];
         хиз := хиз mod размербуф + 1; дес(использов);
         регстатуса[6] := истина; выпвв; регстатуса[6] := ложь;
         послать(неполный)
       end
     end упрэкран;

begin
  хв := 1; хиз := 1;  использов := 0;
  упрэкран
end экран;

device module клавиатура[4];
   define выбратьсим, звонок;
   const размербуф = 32;
   type буфер = array 1 : размербуф of сим;
   var хв, хиз, использов : целый;
         буф : буфер;
         звонок, непустой, неполный : сигнал;

  procedure выбратьсим(var с : сим);
  begin
     if использов = 0 then ждать(непустой) end;
     с := буф[хиз]; хиз := хиз mod размербуф + 1;
     дес(использов);
     послать(неполный);
   end выбратьсим;

   process пруправ[60B];
     var регстатуса[177560B]: биты;
           регданных[177562B] : сим;
         си : сим;
   begin
       loop
          if использов = размербуф then ждать(неполный) end;
          регстатуса[6];= истина; выпвв; регстатуса[6]:=ложь;
          си :== сим(целый(регданных) mod 128);
          if си = 7с then
             послсть(звонок)
          else
             буф[хв] := си; хв := хв  mod размербуф  + 1;
            приращ(использов);
            послать(непустой)
          end
       end
    end пруправ;

begin
 хв := 1; хиз := 1; использов := 0;
 упрклв
end клавиатура;
```

12.3.5. Интерфейс с оператором

Последней компонентой программы управления является операторский интерфейс. Так как это активная компонента, она реализуется в виде процесса, имеющего следующий базовый вид:

```
process оператор;
begin
 loop
   ждать(звонок);
   запрос; (* получить исключительный доступ к консоли *)
   команда чтения;
   команда выполнения;
   освободить (* освободить консоль для других пользователей *)
  end
end;
```

Подробный код для операторского процесса лишется достаточно просто и имеет следующий вид:

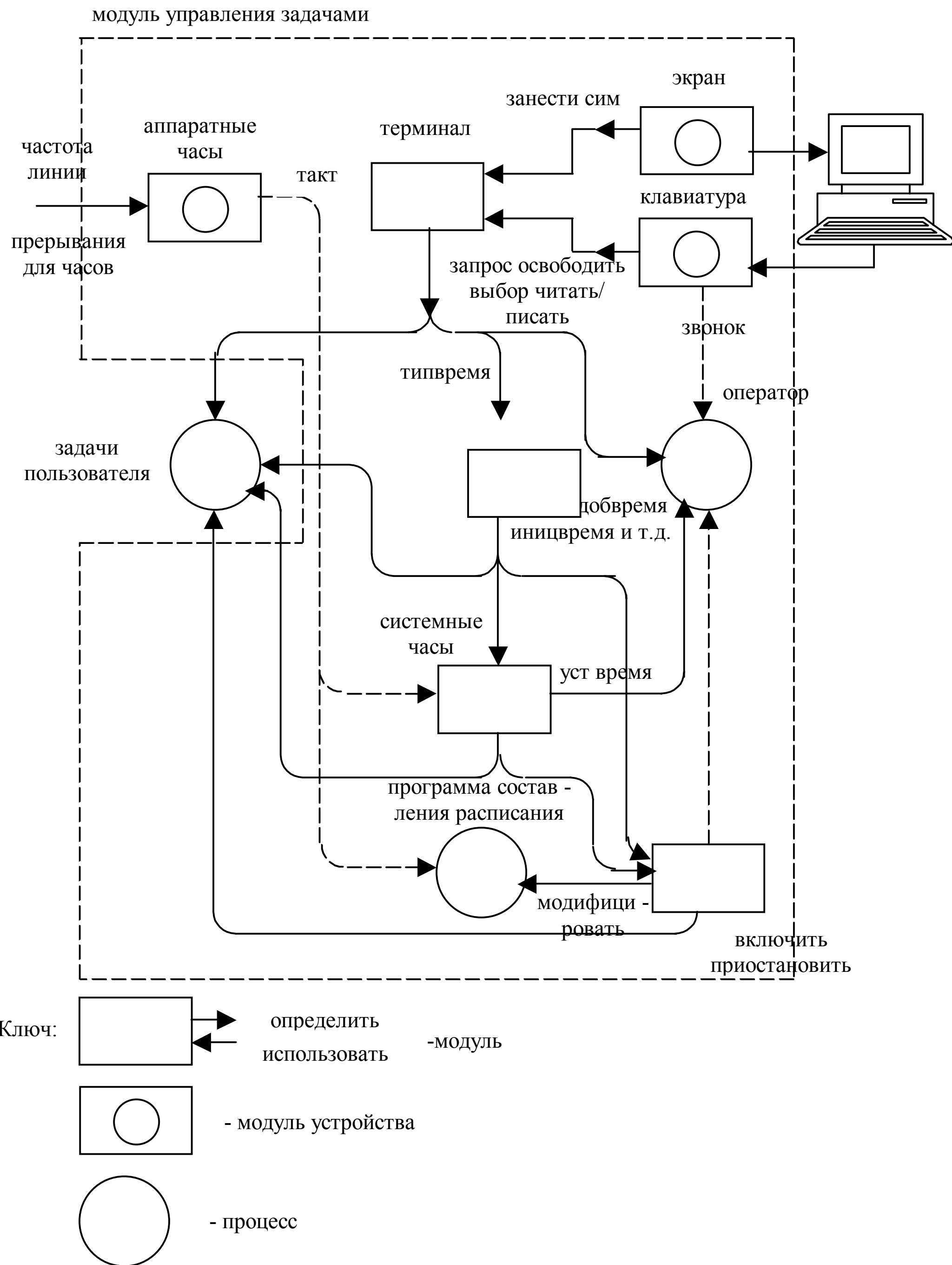
```
process оператор;
  const cr= 15c;
  var команда: сим;
       идзадачи : целый;
      ок: логический;

 procedure ошибка;
    var с : сим;
  begin
   while not кнцстр do читатьсим(с) end;
    писатьстр(‘команда ошибки’); писать(cr)
 end ошибка;

  procedure братькмд(var кмд : сим; var ок : логический);
    var с: сим;
  begin
     писатьсим(cr); писатьстр( ‘ввести команду :’);
    читатьсим(с);
    repeat читатьсим(кмд) until кмд < >’’;
    ок :=( кмд = ’Q’) or (кмд = ‘T’) or (кмд=’S’)
   end выбратькмд;

   procedure старткмд(var ок : логический);
      var период : время;
   begin
     читатьцел(идзадачи, ок);
     if ок then
        читатьвремя(период, ок);
        if ок then  старт(идзадачи, период, ок)
       end
      end
    end старткмд;
```

```
    procedure стопкмд(var ок : логический);
    begin
      читатьцел(идзадачи, ок);
      if ок then стоп(идзадачи, ок) end
    end стопкмд;

    procedure кмдвремя(var ок : логический);
       var t : время;
    begin
      читатьвремя(t, ок);
      if ок then установвремя(t) end
    end кмдвремя;

begin (* оператор *)
  loop
    ждать(звонок);
    запрос;
    выбратькмд(команда, ок);
    if ок then
       case командна of
          'T': begin кмдвремя(ок) end;
          'S': begin старткмд(ок) end;
          'Q': begin стопкмд(ок) end
       end
     end;
     if ок then
       писать('Команда принята'); писатьсим(cr)
     else
       ошибка
     end;
    освободить
  end
end оператор;
```

12.3.6. Программа управления задачами

Построив каждую из компонент программы управления задачами, можно теперь интегрировать их в один модуль с одним хорошо определенным интерфейсом с задачами пользователя следующим образом:

```
module прогупрзадач;
   define запрос, освободить, читатьсим, (* обслуживание
                                               терминалов *)
     писатьсим, читатьцел, писатьцел,
```



процесс Рис. 12.2. Структура программы управления задачами.

```
    писатьстр, кнцстр,
    время, добвремя, читатьвремя,     (* тип время *)
    писатьвремя,
    выбратьвремя                      (* системное время *)
    включить, приостановить;          (* команды
                                      составления расписания *)
  device module экран[4]; ...
  device module клавиатура[4]; ...
  interface module терминал; ...
  module типвремя; ...
  device module аппчасы[6];
  interface module системчасы; ...
  interface module табзадач; ...
  process прогсосрас; ...
  process оператор; ...
  begin
    прогсосрас; оператор
  end прогупрзадач;
```

Заметьте, что порядок, в котором описаны модули в программе управления задачами, выбран таким, чтобы обеспечить описание всех идентификаторов перед их использованием. На рис. 12.2 изображена система, которую мы построили; она может помочь уяснить взаимосвязь между компонентами.

И наконец, нужно добавить несколько слов о тестировании и отладке. Предоставляемая Модулой модульная структура программы позволяет выполнять отладку четко определенным способом. Система строится последовательным наращиванием по одному модулю, начиная с модулей самого низкого уровня: модулей устройств, затем добавляются по очереди модули более высоких уровней. Каждый модуль тщательно проверяется перед добавлением следующего модуля, пока не будет построена вся система. Ключевым моментом здесь является то, что модуль дает очень надежный интерфейс между своими внутренними частями и внешней средой. В результате после завершения исчерпывающей отладки модуля введение нового модуля с ошибками не может, как правило, нарушить его правильное функционирование. Поэтому при добавлении каждого нового модуля любые появляющиеся ошибки можно считать безусловно связанными с этим новым модулем и ни с каким другим. Последующее исправление ошибки, таким образом, сильно упрощается.

12.4. РЕАЛИЗАЦИЯ МОДУЛЫ

В спецификациях языка Модула, разумеется, не дается четких определений того, как нужно реализовывать язык. Но, впрочем, при разработке большинства его параллельных особенно-

стей имелась в виду некоторая конкретная стратегия реализации (Вирт, 1977с). Поэтому сам язык можно понять значительно лучше, если разобраться в этой стратегии.

При разработке Модулы авторы прежде всего заботились о том, чтобы язык был способен конкурировать, с точки зрения программирования, с языками ассемблера не только в отношении выразительных возможностей, но и с точки зрения эффективности. В связи с этим правила языка специально были вы-

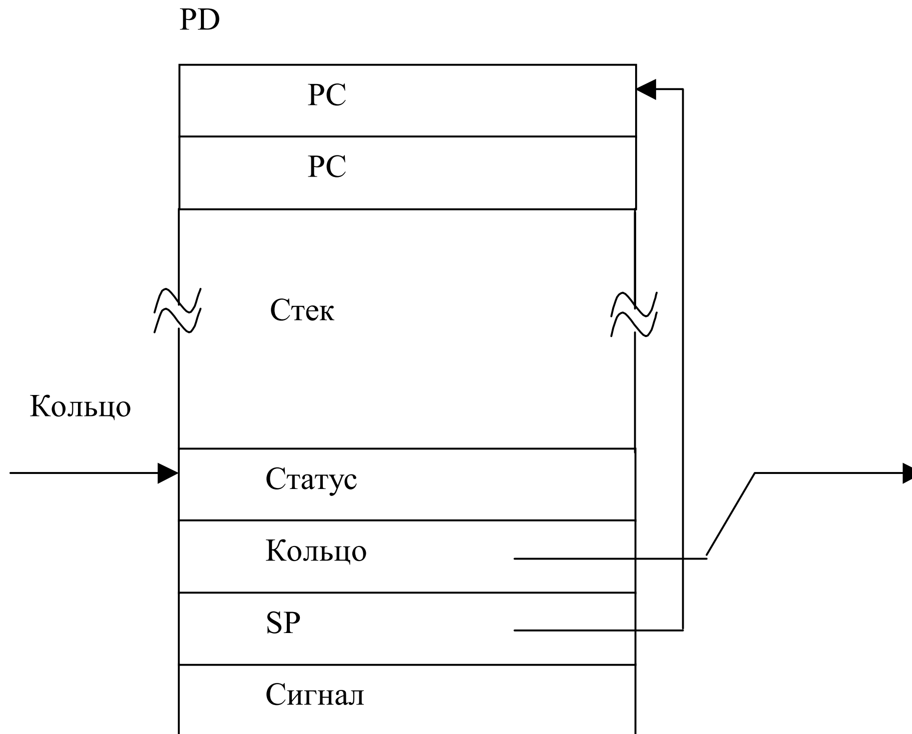


Рис. 12.3. Структура описателя процесса (PD), в Модуле.

браны так, чтобы имелась возможность такой организации процесса на однопроцессорных машинах, при которой дополнительные расходы для прогона задачи минимальны.

Каждая активация процесса в написанной на Модуле программе представляется описателем процесса (PD). Каждый PD содержит достаточно информации для возобновления процесса, который он представляет, при очередных доступах к процессору. Типичный PD приведен схематически на рисунке 12.3. Этот PD содержит четыре фиксированных поля и стек для локальной рабочей памяти процесса. Для каждого PD память распределяется динамически при выполнении соответствующего оператора вызова процесса. Впрочем, предусмотренное в языке ограничение, что операции вызова процессов могут производиться только на самом внешнем уровне программы.

означает, что может быть использована простая стратегия последовательного распределения.

Когда процесс приостанавливается, счетчик текущей программы (PC) и статусное слово процессора (PS) запоминаются в вершине стека, а указатель на вершину стека запоминается в поле SP описателя процесса. Поле статуса PD фиксирует статус процесса на время приостановки. Значение 0 указывает, что процесс готов к выполнению, а значение, большее 0, указывает на ранг задержки. Все описатели процессов связываются вместе в кольцо с использованием поля кольца в PD, и специальный регистр с именем CP всегда указывает на текущий

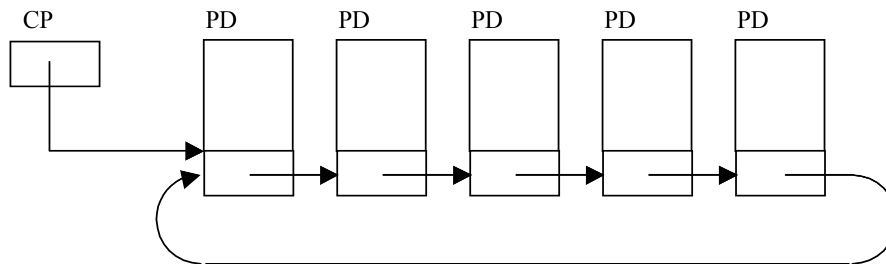


Рис. 12.4. Кольцо описателей процессов в Модуле.

активный процесс в кольце так, как показано на рис. 12.4. Сигнальное поле PD используется для связывания всех процессов, ожидающих данного сигнала, в список; сама сигнальная переменная содержит указатель на начало этого списка.

Стратегия составления расписания, распределяющего физический процессор среди различных процессов Модулы, чрезвычайно проста. Контекстные переключения происходят лишь тогда, когда текущий процесс выполняет операторы *послать* и *ждать*. Конкретно при выполнении каждого из этих операторов происходит следующее:

(i) *послать* (*s*) — запомнить PC и PS в стеке, запомнить указатель на стек в поле SP, задать статусу значение, равное 0 (т. е. готов). Развязать PD, на который указывает 5, и установить регистр CP так, чтобы он указывал на этот PD. Наконец, перевести процессор на этот процесс, загрузив регистры SP, PC и PS по значениям из PD.

(ii) *ждать* (*s, n*)— запомнить PC и PS в стеке, записать указатель на стек в поле SP, установить статус равным *n*. Связать PD с упорядоченным списком для сигнала *s*, используя в качестве ключа *n*. Идти с CP по кольцу, пока не будет найден PD, у которого статус равен 0; затем возобновить процесс по пункту (i).

Из этой зависящей от событий стратегии составления расписания следует, что контекстные переключения происходят только в известных точках выполнения каждого процесса, т. е. в операторах *ждать* и *послать*. Отсюда мы имеем два важных следствия:

(a) Компилятор может обеспечить, чтобы всегда при выполнении операторов *послать* и *ждать* не использовался *ни* один из рабочих регистров физического процессора. Поэтому контекстные переключения не требуют запоминания и восстановления рабочих регистров процессора, что способствует высокой эффективности.

(b) Внутри интерфейсного модуля контекстное переключение может произойти только в операторах *ждать* и *послать,* но эти операторы не подчиняются правилу взаимного исключения. Поэтому не требуется никакого явного механизма для обеспечения взаимно исключающего доступа к интерфейсным модулям.

Теперь должно стать понятным, почему семантика операций, *ждать* и *послать* внутри интерфейсных модулей не следует обычному правилу, согласно которому правилу взаимного исключения не подчиняется только оператор *ждать*. Если выполнение оператора *послать* не влечет за собой контекстное переключение, то число событий, приводящих к обращению к программе расписания, значительно уменьшается; в связи с этим может существенно увеличиться вероятность монополизации процессами процессора. Но из определения оператора *послать в* языке явно следует, что контекстное переключение должно иметь место. Поэтому при условии, что каждый циклический процесс выполняет по крайней мере один оператор *ждать* или *послать* в своем цикле, кольцевая структура обеспечивает, что ни один процесс не будет лишен на долгое время процессора.

До сих пор рассматривались только обычные процессы. Ясно, что процессы устройств не могут обрабатываться описанным способом, так как они управляются прерываниями от связанного с ними машинного оборудования ввода-вывода. Поэтому в отличие от обычных процессов процессы устройств требуют существенного параллелизма.

Процессы устройств представляются PD обычным способом; однако PD устройств не связываются в кольцо. Вместо этого запоминается указатель на PD каждого устройства в соответствии с каждым соответствующим ему прерыванием. Процесс

устройства может быть в одном из трех состояний: ожидание сигнала, выполнение или ожидание завершения оператора *выпвв* (т. е. появление прерывания). Ожидающие сигнала процессы устройств объединяются в связанный список, соответствующий этому сигналу, точно так же, как и в случае обычных процессов. Другие два состояния обрабатываются специальным способом; чтобы понять, как это делается, рассмотрим ситуа-

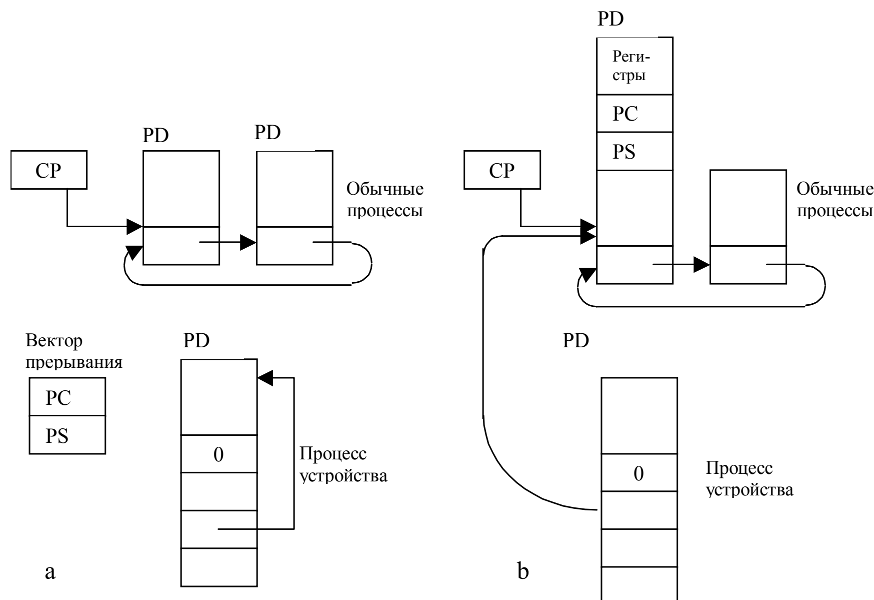


Рис. 12.5. Работа процесса устройства, (а) Процесс устройства обрабатывает операцию *выпев.* (Ь) Процесс устройства выполняется.

цию, изображенную на рис. 12.5. В этом примере система состоит из двух обычных процессов и одного процесса устройства. В части (а) процесс устройства выполняет эператор *выпвв* и, следовательно, находится в неактивном состоянии. Его регистры PC и PS хранятся в виде вектора в ячейках связанного с ним прерывания. При появлении прерывания регистры PC и PS текущего выполняемого процесса автоматически записываются в его стек и в процессор загружаются аппаратно новые значения PC и PS из ячеек вектора прерывания. Затем процесс устройства запасает рабочие регистры процессора, которые использовались прерванным процессом, в стеке процессора, загружает свой собственный SP из своего PD и в заключение использует свое поле кольца для связывания PD прерванного процесса (в соответствии с определенным значением CP). Получившаяся ситуация изображена на рис. 12.5(b). Оператор *выпев* и процесс устройства продолжают выполняться до тех пор, пока не появится новая операция *ждать* или *выпев.* Когда

это происходит, процесс, который был прерван, возобновляется, восстанавливая свои регистры с вершины стека. Эта ситуация возвращается к случаю, приведенному в части (а) рисунка. Важно отметить, что прерванный процесс мог бы быть процессом устройства с низким приоритетом, а не обычным процессом, как в примере. Это соответствовало бы обработке вложенных прерываний.

Главным отличием между контекстными переключениями обычных процессов и переключениями, относящимися к процессам устройств, является то, что первые процессы синхронны и, следовательно, предсказуемы, в то время как процессы во втором случае асинхронны и соответственно непредсказуемы. Как было отмечено ранее, синхронные контекстные переключения не требуют сохранения регистров и не нужно никакого явного механизма для обеспечения взаимно исключающего доступа к интерфейсным модулям. Асинхронное же переключение требует переписи регистров, что видно из приведенного выше примера. Кроме того, на первый взгляд кажется, что асинхронное переключение требует специального механизма для гарантирования взаимного исключения. На самом деле в Модуле никакой такой механизм не нужен ни для интерфейсных модулей, ни для модулей устройств. Это может быть показано следующим образом.

Во-первых, рассмотрим интерфейсные модули. Если прерывание происходит во время выполнения процессом P интерфейсной процедуры, то он будет прерван и будет возобновлен подходящий процесс устройства D. Однако целостность модуля по-прежнему поддерживается, так как механизм обработки процесса устройства гарантирует, что P будет возобновлен прежде любого другого обычного процесса. Единственным процессом, который мог бы, возможно, получить доступ к интерфейсному модулю прежде, чем будет возобновлен прерванный процесс, и, следовательно, нарушить правило взаимного исключения, является процесс устройства D. Однако ограничение языка, состоящее в том, что процессы устройств не могут вызывать нелокальные процедуры, гарантирует, что такое произойти не может.

Во-вторых, рассмотрим модули устройств. Весь код в модуле устройства выполняется со специфицированным приоритетом. Этот приоритет всегда следует выбирать так, чтобы он был достаточно высок для подавления прерывания, связанного с процессом устройства внутри этого модуля. Поэтому, когда обычный процесс выполняет интерфейсную процедуру внутри модуля устройства, он не может быть прерван процессом устройства внутри того же самого модуля. Следовательно, модуль устройства гарантирует взаимно исключающий доступ к своим

интерфейсным процедурам простой регулировкой приоритетом процессорного прерывания.

Из данного общего описания того, как можно реализовать Модулу на однопроцессорной машине (т. е. на PDP 11), должно быть понятным, как тщательно проект языка был подогнан для обеспечения небольших издержек во время прогона программы в терминах как времени выполнения, так и требований к памяти для подпрограмм *послать, ждать* и *выпвв.* Полная система поддержки прогона программ требует для PDP 11 фактически всего 150 слов кода. В результате написанные на Модуле программы могут конкурировать с программами, написанными на языке ассемблера, с точки зрения как использования памяти, так и скорости даже на небольших целевых машинах. Впрочем, эффективность Модулы получается за счет некоторой потери общности. Это один из аспектов, обсуждаемых в следующем разделе.

12.5. ОБСУЖДЕНИЕ

Как было сказано ранее, Модула разрабатывалась для очень специальной сферы приложения — для небольших встроенных вычислительных систем. С целью удовлетворить требованиям этой области очень большое внимание уделено обеспечению того, чтобы язык был достаточно мал, допускал простую и эффективную компиляцию и чтобы также можно было генерировать очень компактный и эффективный код. Одновременно с этим, разумеется, язык должен обладать достаточной выразительной силой, удовлетворяя нужды своих пользователей.

То, что Модула небольшой и эффективный язык, не вызывает сомнения. Описанный Холденом и Вандом (1980) компилятор требует всего лишь 16К слов памяти PDP 11 и компилирует n строк программы примерно за $(5 + n)/12$ секунд на PDP 11/40. Кроме того, язык очень удобен на практике. Он использовался при реализации ряда достаточно сложных систем с большим успехом (см., например, Эндрюс, 1979, Рунсиман, 1980). Тем не менее компактность языка означает, что многие из особенностей, описанные в первой части этой книги, не могли быть включены в описание языка. Кроме того, при разработке некоторых из включенных в язык особенностей пришлось пойти на существенные ограничения для достижения максимальной эффективности.

Из того, что не было включено в язык, наиболее существенным является следующее:

— тип вещественных данных
— механизм обработки ошибок

— механизм подтипизации для скалярных типов
— оператор цикла
— вариантные записи

Из этих недостатков действительно серьезными являются только первые два. Многие приложения реального времени требуют работы с фиксированной и плавающей запятой, поэтому ясно, что некоторая форма вещественного типа данных необходима. Отсутствие явного механизма обработки ошибок ведет к созданию не очень ясного кода даже в простейших случаях (ср. процедуру *читатьвремя* в модуле *типвремя* из разд. 12.3). Включение оператора **goto** было бы простым и эффективным средством увеличения возможностей языка в этом направлении без существенного увеличения его сложности. Опущение остальных трех возможностей в действительности не очень ограничивает выразительную силу языка, но может оказаться неожиданно неприятным.

Основная критика Модулы касается проекта его возможностей мультиобработки и программирования устройств низкого уровня. Все процессы в Модуле анонимны, т. е. у процесса нет никакой возможности сослаться на другой процесс. Как следствие, в Модуле нельзя написать супервизорную программу составления расписания, когда центральный супервизорный процесс полностью управляет рядом процессов *задач.* Вместо этого должна использоваться схема, аналогичная схеме из предыдущего примера программы (разд. 12.3), где каждый процесс задачи должен приостанавливать себя сам. Это означает, что Модулу нельзя использовать в системах, в которых возможно неправильное функционирование процесса. В частности, Модулу нельзя применять при реализации операционных систем общего назначения. Простого решения этой проблемы не существует, так как она является прямым следствием зависящей от событий стратегии составления расписания. Вандом (1980) был предложен механизм, обеспечивающий управление выполнением процесса в Модуле, но он связан с расширением языка. Эту проблему разработчики сочли столь серьезной, что в Модуле 2 от процессов отказались вовсе в пользу взаимодействующих подпрограмм (Вирт, 1980); однако такая модификация вызвала изменение природы языка и области его применения, и Модула 2 является в большей степени языком реализации систем, чем языком программирования реального времени.

Вторым следствием этой зависящей от событий стратегии составления расписания является то, как пришлось модифицировать в интерфейсных модулях традиционную семантику сигнала *послать.* Хотя это не оказывает серьезного влияния на применяемость языка, эта модификация затрудняет доказатель-

ство правильности функционирования интерфейсных модулей; а различие трактовки интерфейсных модулей и модулей устройств эстетически неприятно.

В области программирования низкого уровня у Модулы имеются три недостатка. Во-первых, на регистры устройства можно отображать только скалярные типы данных. Это очень сильно затрудняет программирование регистров, которые разделены на поля, как в примере таймера Intel 8253 из разд. 7.2.2. В таких случаях сильная типизация является скорее помехой, чем средством поддержания, так как единственным простым способом обращения к каждому полю является использование операций and и or для маскирования и объединения целых типов. Конечно, адекватным решением было бы разрешение отображать комбинированные типы на регистры устройств наряду со скалярными типами. Отсюда, следовательно, нужно сделать следующий полезный вывод: сильная типизация полезна только тогда, когда пользователю предоставляется достаточно богатое множество типов данных, которые удовлетворяют все его нужды, а не только некоторые из них.

Во-вторых, Модула не позволяет процессам налаживать связь с прерываниями машинного оборудования через машинные сигналы так, как было описано в разд. 7.3.3. Вместо этого имеется более ограниченный интерфейс, предоставляемый оператором *выпвв.* В результате обычные процессы должны всегда связываться с оборудованием ввода-вывода с помощью процесса устройства. Для буферизованной передачи одного символа, такой, как в модулях *экран* и *клавиатура* из разд. 12.3.4, оператор *выпвв* вполне адекватен. Но для передачи DMA, такой, как в примере модуля *читать-диск* из разд. 7.4.2, он мало пригоден из-за больших издержек. Вместо того чтобы заставлять вызывающий процесс ждать прямого появления сигнала прерывания, необходимо ввести процесс устройства, который всего лишь преобразует прерывание в сигнал программного обеспечения, а тот затем передается вызывающему процессу.

И последнее, процессам устройств не разрешается обращаться к нелокальным процедурам. В результате процесс устройства не может работать с абстрактным типом. Серьезность этой проблемы хорошо видна на примере программы, в которой процесс управления *аппчасы* (разд. 12.3.3) должен бы увеличивать системное время каждую секунду. Но системное время представлено абстрактным типом, который может обрабатываться только с помощью операций, предоставляемых модулем *типвремя.* Однако эти операции недоступны процессу программы управления, поэтому приходится вводить второй процесс часов для работы с системным временем.

Несмотря на перечисленные выше недостатки, Модула является тем не менее очень хорошим примером проекта языка программирования. Если предоставляемые Модулой возможности сравнить с теми, которые были предложены в разд. 9.7, то можно увидеть, что они очень хорошо соответствуют спецификациям языка реального времени для сильно ограниченной области приложения. Язык Модула, имеющий несколько экспериментальный характер, может быть без труда надстроен до уровня практического индустриального языка с помощью небольшого числа непринципиальных добавлений: вещественные числа, оператор **goto,** отображение записей на регистры устройств, оператор цикла **for** и конструкторы запись/массив. Впрочем, маловероятно, чтобы проводилась работа по усовершенствованию Модулы, так как этот язык отошел в сторону в связи с напряженной деятельностью, касающейся языка Ада. Поэтому есть основания считать, что Модулу будут помнить главным образом как предшественника Ады в двух основных аспектах: во-первых, в том, что модули являются простым и эффективным механизмом абстракции данных, и, во-вторых, в том, что устройства низкого уровня можно программировать непосредственно из языка реального времени высокого уровня. Хотя накопленный при работе с Модулой опыт был хорошо использован разработчиками Ады, все же жаль, что Модула потерял всеобщее признание как очень хороший язык реального времени сам по себе. В рамках очень компактной схемы Модула позволяет программировать небольшие встроенные вычислительные системы непосредственно в надежном языке высокого уровня без какого-либо обращения к программированию на языке ассемблера. Учитывая еще и то, что Модулу очень легко изучать, общая полезность этого языка, особенно в области встроенных микропроцессорных систем, должна быть вполне очевидна.

Глава 13. АДА

13.1. ПРЕДПОСЫЛКИ

Язык программирования Ада разработан по заказу министерства обороны США. Главным стимулом его создания было осознание того факта, что стоимость программного обеспечения в области встроенных систем растет со все возрастающей скоростью. Более того, расходы возрастают не только на этапах разработки новых систем, но и при периодичзских затратах по поддержанию существующих систем. Основной причиной этих

высоких издержек было признано отсутствие стандартизации. И на самом деле, проведенный в свое время министерством обороны подсчет показал, что в области встроенных систем активно используется более 350 языков. Поэтому было решено, что необходимо создание единственного стандартного языка.

Первым шагом явилось опубликование в 1976 году набора требований к стандартному языку программирования реального времени в форме документа, называемого Тинман («Оловянный»). Далее были предприняты усилия по оценке того, удовлетворяет ли какой-либо из существующих языков этим требованиям. Результаты этих оценок показали, что ни один из существующих языков не удовлетворяет полностью условиям Тинман, но три языка — Паскаль, PL/1 и Алгол 68 — обладают достаточно здравой и испытанной структурой, чтобы быть положенными в основу проекта нового языка.

Второй этап предварительных исследований начался в январе 1977 года с пересмотра документа Тинман; новые, более точные и подробные требования к языку были изложены в документе, называемом Айронман («Железный»). Ряду организаций было предложено разработать проект нового языка на базе одного из языков: Паскаля, PL/1 или Алгола 68. Из семнадцати предложений было отобрано четыре, которые и должны были определить окончательный проект. Каждый из этих четырех языков базировался на Паскале, и каждому было дано условное название, обозначающее один из цветов с целью сохранить анонимность их создателей (чтобы обеспечить беспристрастность при оценке языков). Окончательный проект языка Айронман был опубликован в феврале 1978 года. От проектов Желтый и Голубой (представленных соответственно фирмами SRI International и Softech) отказались окончательно, а проекты Красный и Зеленый (представленные фирмами Intermetrics и CII Honeywell Bull соответственно) были отобраны для дальнейшего уточнения.

Во время третьего этапа подверглись пересмотру проекты Красный и Зеленый, чтобы удовлетворить новым условиям документа с названием Стилман («Стальной»). Проектные требования Стилман опубликованы в марте 1979 года, а в мае 1979 года был отобран язык Зеленый, которому суждено было впоследствии стать языком Ада.

Четвертый и последний этап состоял из тщательно разработанных проверок и оценок. На основе полученного опыта в язык ввели ряд изменений, после чего в июле 1980 года были опубликованы спецификации языка Ада. Хотя и можно ожидать, что, прежде чем язык станет официальным международным стандартом, в него будут внесены еще некоторые исправления, *маловероятно,* чтобы эти изменения вышли за рамки окончатель-

ной настройки. Описываемый в этой главе язык базируется на июльской версии 1980 года (Ихбиа и др., 1980).

Прежде чем перейти к рассмотрению самого языка Ада, представляется интересным бегло рассмотреть путь его разработки. Во-первых, выбранный министерством обороны США подход — привлечение для разработки проекта конкурирующих организаций — позволил обойти ряд неприятных «узких мест», которые возникают при разработке языка специальным комитетом. В этом отношении Аду следует сравнить с языком CHILL, который был разработан для телекоммуникационных систем под эгидой CCITT (1980). Хотя предполагалось, что CHILL должен предоставлять аналогичные с Адой возможности, это довольно слабый язык, содержащий много неортогональных свойств. Во многих случаях он предоставляет несколько различных механизмов и синтаксических конструкций для выполнения одной и той же функции. Это является прямым следствием того, что язык разрабатывался комитетом, объединяющим антагонистические точки зрения. Во-вторых, вся программа разработки Ады велась по строгому расписанию, соблюдалось точное выполнение сроков. С организационной точки зрения это было великолепно и, на самом деле, без соблюдения этого строгого расписания все мероприятие могло затянуться на годы. Впрочем, когда подошли сроки завершения проектов языков Красный и Зеленый в соответствии с требованиями Стилман, проект языка Красный был еще не завершен. В ретроспективе многие полагают, что потенциально проект языка Красный был лучше. Но по сравнению с проектом требований Айронман он претерпел достаточно радикальные изменения, и во время выбора лучшего проекта не представлялось возможным полностью оценить последствия этих изменений. Поэтому проект Зеленый, который был завершен, представлял выбор «наименьшего риска», и соответственно его взяли за основу. Если бы не требовалось завершить разработку точно к марту 1979 года, то вполне возможно, что Адой стал бы не Зеленый язык, а Красный. Заметим, что многие сторонники Паскаля в том, что все четыре предложения серии Айронман базируются на Паскале, видят ясное доказательство присущего этому языку превосходства. Хотя это может быть и так, выбор Паскаля упомянутыми организациями не следует считать решающим доказательством этого превосходства. Дело в том, что документ Айронман в основном был написан с использованием терминологии Паскаля. Поэтому не удивительно, что все разработчики включили ее в свой язык. В этой связи более нейтрально написанный документ требований, подчеркивающий скорее цели вместо специфических языковых особенностей, привел бы к появлению большего разнообразия в проектах серии Айронман, а эта

большая свобода могла бы реализоваться в лучшем конечном продукте.

Тем не менее, несмотря на приведенные выше комментарии к процессу разработки Ады, конечный результат является, безусловно, мощным, хорошо сконструированным языком, который, вне всякого сомнения, получит широкое распространение во всем мире. Его успех обеспечивается не только потому, что это по существу хороший проект, но также и потому, что он будет частью системы разработки завершенного программного обеспечения. Наряду с компилятором, поддерживающая Аду система предоставит стандартные редакторы, средства обнаружения и исправления ошибок, программы форматного вывода текстов, системы управления библиотеками и т. д. Более того, вся система целиком будет стандартизована, так что может быть гарантирована мобильность как программ, так и сравнительная легкость переучивания программистов (DARPA, 1980).

Дальше в этой главе мы сконцентрируем наше внимание на самом языке Ада. В следующем разделе достаточно подробно описывается язык, затем в разд. 13.3 приводится пример его использования. Раздел 13.4 завершает главу, в нем обсуждаются основные преимущества и слабости языка.

13.2. ЯЗЫК

Ада — это очень большой и сложный язык, его полное описание потребовало бы значительно больше места, чем предоставляет одна глава. Но к счастью, большинство новых идей, которые содержатся в Аде, были уже обсуждены более или менее подробно в первой части этой книги. Поэтому такие понятия, как производные типы, совмещение, исключения и т. д., вводятся без объяснения. Вместо этого мы отсылаем читателя за более подробной информацией к соответствующему разделу в первой части.

13.2.1. Общий стиль

Основной лексический стиль Ады достаточно традиционен, он отражает влияние Паскаля. Идентификаторы составляются из последовательности букв, цифр и подчеркиваний. Ключевые слова зарезервированы и не могут переопределяться, хотя такие предопределенные идентификаторы, как *целый,* переопределяться могут. При текстуальном представлении программы, написанной на языке Ада, предпочтительнее писать зарезервированные слова строчными символами, а идентификаторы — прописными. Но в этой книге и те и другие будут писаться строчными буквами, зарезервированные слова будут выделяться

жирным шрифтом для сохранения единообразия стиля во всей книге. Числовые литералы — это либо вещественные числа для представления значений с плавающей и фиксированной запятой, либо целые числа. Использование основания системы счисления, отличающегося от 10, обозначается заключением числа в символы #, перед первым из которых стоит основание. При записи символов они заключаются в одиночные кавычки, а строки заключаются в двойные кавычки. Наконец, комментарии отмечаются двойным дефисом и продолжаются до конца строки. Следующие примеры иллюстрируют форму различных лексических элементов:

*x следу ющее_значение j*31 -- *идентификаторы*
with rem procedure --*зарезервированные слова*
+ /=; --*операции и ограничители*
0 1234 --*целые литералы с основанием* 10
2#1010# 16#*FF*# --*целые литералы с указанием основания*
1.34 0.056 --*вещественные литералы*
1.0*E*6 2.4*E* - 10 --*вещественные литералы с экспонентой*
‘A’ ‘+‘ --*символьные литералы*
“ТЕКСТ” “A” --*строковые литералы*

Директивы компилятору могут быть включены в программу с помощью ключевого слова **pragma** (указание), за которым следует имя указания и соответствующие аргументы, например

pragma *список*(*выбросить*);

задает компилятору команду прекратить последующую выдачу листинга.

Программа на языке Ада состоит из одной или более компилируемых компонент. Каждая компонента — это либо подпрограмма, либо пакет (название модуля в Аде). Перед каждой компилируемой компонентой находится спецификация контекста, в которой указываются имена всех других компонент, от которых данная компонента зависит. Каждая программная среда Ады обеспечивает пакет с именем *стандартный,* который содержит определения предопределенных типов и операций над ними (например, *целый,* +, —, и т. д.). Пакет *стандартный* имеется всегда, и в спецификациях контекста его имя упоминать не обязательно. Кроме этого, обычно будет предоставляться множество стандартных библиотечных пакетов для ввода-вывода, математических функций и т. п. При желании их использовать их имена должны быть указаны в спецификациях контекста.

Для главной программы в Аде нет стандартного обозначения. Впрочем, как правило, для представления главной программы используется процедура без параметров, идентифицируемая включением указания *главный.* Итак, примером полной, хотя и несколько тривиальной программы на языке Ада может быть следующая запись:

```
with текст_вв, вещ_операции;     --библиотечные пакеты
use вещ_операции;                --определяют тип вещественный
                                 --и пакеты вещ_вв
                                 - -и вещ_функции
procedure площадь_треугольника is
   pragma главный;
   площадь, s, a, b, c: вещественный;
   use вещ_вв,                   --определяет выбрать и занести
                                 --для вещественных
       текст_вв;                 --определяет занести для строк
                                 --и новой_строки.
     вещ_функции;                --определяет корень
begin
   занести("Ввести стороны треугольника");
   выбрать(a); выбрать(b); выбрать(c);
   s := 0.5 * (a + b + c);
   площадь := корень(s * (s - a) * (s - b) * (s - c));
   занести("Площадь равна"); занести(площадь);
   новая_строка;
end площадь_треугольника;
```

Предложение **with** обозначает спецификацию контекста. В нашем случае она указывает, что в следующей компилируемой компоненте будут использоваться пакеты *текст_вв* и *вещ_операции.* Различные предложения **use** делают имена объектов непосредственно доступными в этих пакетах (см. разд. 13.2.5). Структура самой процедуры аналогична процедуре Паскаля, но заметьте, что точка с запятой используется в качестве ограничителя, а не разделителя; поэтому она не может быть опущена перед закрывающим ключевым словом, таким, как end.

13.2.2. Простые типы данных

Языку Ада свойственна система сильной типизации, базирующаяся на именной эквивалентности, такой, как описанная в разд. 2.2. Каждое описание типа вводит новый тип, который отличается от всех других типов. Диапазон разрешенных для каждого типа значений может быть ограничен спецификацией подходящих ограничений либо в самом описании типа, либо в описании объекта данного типа, либо в описании подтипа,

Рассмотрим, например, следующие описания (заметьте, что при введении описаний переменных не используется никаких явных ключевых слов типа **var**):

type *индекс* **is new** *целый* **range** 0..100;
subtype *мал_цел* **is** *целый* **range -**128.. 127;
знач: *целый*;
i, j: *мал_цел*;
k : *целый* **range** 0.. 127;
инд : *индекс*;

Переменные *знач, j, i, k* имеют тип *целый.* Диапазон допустимых значений для *знач* зависит от реализации: для *i* и *j* специфицируется описанием подтипа для *мал_цел,* а для переменной *k* специфицируется в ее описании. Так как эти переменные одного и того же типа, они могут встречаться в выражениях в любом порядке, например, выражение

i := *знач* * *j* + *k*;

законно. Если вычисленное значение выражения находится вне диапазона, специфицированного для *i,* то вызывается исключение с именем *ошибка_ограничения.* Тип индекс является новым типом, производным от типа *целый.* Поэтому переменная *инд* не может употребляться в операциях с другими переменными, например, выражение

инд := *знач* + *k*;

незаконно. Чтобы выполнить такую операцию, необходимо применить операцию явного преобразования типа, т. е.

инд := *индекс*(*знач* + *k*);

Производный тип обычно наследует все литеральные значения, определенные для его родительского типа. Кроме этого, если его родительский тип является предопределенным, то он также наследует все предопределенные для этого типа операции. Если родительский тип определен внутри пакета (см. 13.2.5), то определенные в этом пакете операции наследуются производным типом, в противном случае никаких операций не наследуется. Операции отношения = и /= (не равно) и операция присваивания предоставляются всем типам (если у них не было спецификации *ограниченный* — см. 13.2.5).

Любой переменной может быть задано начальное значение одновременно с ее описанием, например

i: *мал_цел* :-= 0;

описывает переменную *i* и задает ей начальное значение, равное 0. Константные объекты вводятся с помощью включения специального ключевого слова, как, например, ниже:

ci : **constant** *мал_цел* := *f*(*i*) +1;

описанные таким образом объекты можно только читать. Этот последний пример показывает также, что начальное значение не обязательно вычисляется как статическое выражение во время компиляции, оно может быть вычислено во время работы программы при обработке описания. Литеральные числовые константы могут вводиться без спецификации их типа, например,

pi: **constant** := 3.14159;
один : **constant :=** 1;

где *pi* принадлежит типу *«универсальный_вещественный»,* а *один* — типу *«универсальный_целый».* Универсальный тип совместим с любым типом, который является производным от типа с той же самой числовой базой, поэтому *знач* := *один*; и *инд* := *один*; законны.

Перечислимые типы описываются в следующих примерах; заметим, что перечислимые литералы могут быть совмещенными (см. 2.3.1).

type *цвет* **is** (*красный, зеленый, желтый, голубой*);
type *свет_светофор* **is** (*красный, янтарный, зеленый*);
type *день* **is** (*пон, вт, среда, чет, пят, суб, вос*);

Все неоднозначности, которые могут встретиться при использовании совмещенных идентификаторов *красный* и *зеленый,* могут быть разрешены с помощью квалифицированных выражений вида

имятипа'(*выраж*)

Поэтому если совмещенная процедура с одним параметром *P* применяется как к объектам типа *цвет,* так и к объектам типа *свет_светофор,* то выражение

P(*красный*)

неоднозначно. Поэтому это обращение должно быть записано в виде

P(*цвет'*(*красный*))

Или

P (*свет_светофор'*(*красный*))

Каждый тип в Аде обладает определенными атрибутами, которые можно узнать в программе с помощью запроса атрибута, имеющего вид

имятипа’атрибут(*возможен параметр*)

В случае когда тип *T* дискретный, *T’первый* возвращает первое значение этого типа, а *T’последний* возвращает последнее значение. Функции предшествования и следования также определяются как запросы атрибутов; отсюда

цвет’след(*зеленый*) = *желтый*

и

свет_светофор’пред(*зеленыи*) = *янтарный*

Типы *логический* и *символьный* являются предопределенными перечислимыми типами, где

type *логический* **is** (*ложь, истина*);

и

type *символьный* is (*...’A’, ‘B’, ‘C’, ... и т. д.*)

Пользователь может ввести свое собственное множество символов с помощью перечисления, например,

type *шестнадзнач* is (’0’, ‘1’, ‘2’, ... ‘*D*’, ‘*E*’, ’*F*’);

Порядковые значения перечислимого типа *T* задаются с помощью *T’поз,* следовательно:

шестнадзнач’поз(‘*D*’) = 13

а обратная операция обеспечивается конструкцией *T’ знач,* например,

шестнадзнач’знач(12) = ‘*C*’

Возможности определения перечислимых типов в Аде позволяют представлять целые величины и величины с плавающей и фиксированной запятой. Построение перечислимых типов таково, что они позволяют писать действительно мобильные программы. Основные идеи лучше всего продемонстрировать на целых типах. Каждая реализация Ады обеспечит предопределенный тип *целый.* Кроме этого она может предоставить один или более добавочных типов с большими или меньшими диапазонами, например *длинный_целый, короткий_целый* и т. д. Диапазоны этих типов будут, конечно, зависеть от системы. Поэтому программа, в которой имеется описание

type *мойцел* **is new** *целый* **range** *низкий..высокий*;

может вызвать обращение к *ошибка_ограничение* при переносе на машину, на которой *высокий* и *низкий* лежат вне реализованного диапазона типа *целый.* В Аде эта проблема разрешается тем, что можно опускать имя типа, т. е. Писать

type *мойцел* **is range** *низкий..высокий*;

Эта форма эквивалентна форме

type *мойцел* **is new** *цел_тип* **range** *низкий..высокий*;

где *цел_тип* выбирается из предопределенных типов *длинныйцелый, целый, короткий_целый* и т. д. автоматически компилятором так, чтобы могла быть обеспечена требуемая точность. Аналогичный механизм применим и для типов с плавающей залятой. Например, объект с описанием

type *мойплав* **is digits** 6;

получит тип *мойплав* из подходящего предопределенного типа с плавающей запятой (*плавающий, длинный-плавающий, короткий-плавающий* и т. д.), сохраняя требуемую точность в 6 знаков. Типы с фиксированной запятой обрабатываются несколько по-другому, так как для них нет предопределенных типов. Поэтому все типы с фиксированной запятой должны быть явно описаны пользователем. Форма описания следующая:

type *мойфикс* **is delta** 0.01 **range** 0.0..10.00;

Стоящее после ключевого слова **delta** число специфицирует абсолютную точность, присутствие ключевых слов **delta и range** обязательно. Фактическое значение абсолютной точности выбирается, как правило, так, чтобы оно было степенью 2; это сразу же дает требуемую точность (полностью базовые понятия определения числовых типов обсуждаются в разд. 2.4).

Сведения о предопределенных операциях для описанных выше простых типов приводятся в табл. 13.1 и 13.2. Они достаточно традиционны, следует лишь отметить несколько моментов. Логические операции **and then и or else** являются специальными формами операций **and** и **or** соответственно. Поясним эту специфику на примере. Если вектор является массивом с нижней границей индекса, равной 0, то вычисление часто встречающегося логического выражения

i > = 0 **and** *вектор*(*i*) = 0

может привести к вызову *ошибка-ограничения, если i* < 0, а компилятор стал вычислять второй операнд первым. Для решения этой трудности в Аде введена специальная форма. Поэтому текст

i > = 0 **and then** *вектор*(*i*) = 0

в этом смысле всегда корректен, так как второй операнд, стоящий после **and then,** вычисляется только в случае истинности первого операнда. Операции принадлежности множеству **in** и **not in** используются для проверки, находится ли значение в данном диапазоне или удовлетворяет ли ограничению подтипа. Например,

63 **in** *мал_цел* = *истина*
4 **not in** 1..10 = *ложь_*

Операции * и / могут применяться к любому типу с фиксированной запятой, давая в результате виртуальный тип, называемый *универсальный_фиксированный* с произвольно маленькой

Таблица 13.1. **Одноместные операции**

Операция	Операнд	Результат	Приоритет
+	числовой	то же	4
-	числовой	то же	4
not	логический	то же	4

абсолютной точностью. Вслед за этой операцией пользователь обязан немедленно применять к результату операцию преобразования числового типа, переводя его либо в конкретный тип с фиксированной запятой, либо в целый, либо в тип с плавающей запятой. Целое деление всегда округляет в сторону нуля. Операция rem дает остаток после целого деления так, что

x = (*x/y*) * *y* + (*x* **rem** *y*)

где знак *x* **rem** *y* равен знаку *x.* Операция **mod** вычисляет модуль значения, поэтому

x = *y* * *n* + (*x* **mod** *y*)

для некоторого значения *n.* Следовательно, если делитель положителен, то **rem** и **mod** эквивалентны, например,

13 **rem** 5 = 3 и 13 **mod** 5 = 3

но

13 **rem** - 5 = 3 и 13 **mod** - 5 = - 2

(см. разд. 2.4.1, где обсуждаются проблемы семантики операций целого деления и остатка).

Таблица 13.2. **Двуместные операции**

Операция	Операнды	Результат	Прио-
and	логический	логический	1
Or	логический	логический	1
xor	логический	логический	1
and then	логический	логический	1
or else	логический	логический	1
=	любой тип	логический	2
/=	любой тип	логический	2
<	любой скалярный тип	логический	2
>	любой скалярный тип	логический	2
<=	любой скалярный тип	логический	2
> =	любой скалярный тип	логический	2
In	значение диапазон, подтип	логический	2
not in	значение или ограничени	логический	2
+	числовой	тот же	3
-	числовой	тот же	3
*	целый целый	тот же целый	б
	фикс целый	тот же фикс	
	целый фикс	тот же фикс	
	фикс фикс	универсальный фикс	
	плавающий плавающий	тот же плавающий	
/	целый целый	тот же целый	5
	фикс целый	тот же фикс	
	фикс фикс	универсальный фикс	
	плавающий плавающий	тот же плавающий	
mod	целый целый	тот же целый	5
rem	целый целый	тот же целый	5
**	целый полож. целый	тот же целый	6
	плавающий целый	тот же плавающий	

13.2.3. Структурные типы данных

Ада обеспечивает две формы структурных типов данных, массивы и записи. Множественных типов нет. Регулярные типы описываются спецификацией типа индексов и типов компонент, диапазон типа каждого индекса может не специфицироваться. Основная форма описания регулярного типа иллюстрируется следующими примерами:

type *вектор* **is array** (*целый* **range** < >) **of** *целый*;
type *матрица* **is array** (*целый* **range** < >, *целый*
range < >) **of** *вещественный*;

Типы *вектор* и *матрица* называются регулярными типами без ограничений. Чтобы описать объект такого типа, нужно специфицировать фактические границы индексов. Это можно сделать одним из трех способов, либо введением подтипа, как в

subtype *мойвек* **is** *вектор*(1..10);
*век*1*, век*2; *мойвек*;

либо спецификацией ограничения на индекс, как в :

вектор(1..100);

либо в случае постоянных регулярных объектов границы могут быть выведены неявно по начальным значениям, как в

квек : **constant** *вектор* := (0, 1, 2, 3, 4, 5);

Когда требуется в точности один регулярный подтип, допускается сокращенная форма, например

type *мойвек* **is array** (1..10) **of** *целый*;

что эквивалентно приведенным выше описаниям для *вектор* и *мойвек,* но в этом случае неограниченный базовый тип анонимен.

Массивы и вырезки из массивов могут присваиваться друг другу. Например:

*век*1:= *век2*; --*присваивание массива целиком*
*век*1 := *век*3(10..19) --*присваивание вырезки массива*

Элементы в каждой из частей оператора присваивания должны иметь одинаковое число компонент, но во втором примере индексы не обязательно должны совпадать. Кроме присваивания к одномерным логическим массивам могут применяться логические операции **not**, **or**, **and** и **xor** для выполнения специфицируемой операции над каждой из соответствующих компонент. Разумеется, к массивам любой размерности могут применяться операции отношения = и /=, но к одномерным дискретным массивам могут применяться также операции сравнения <, >, <=, >= для проверки лексикографического упорядочения. Наконец, для одномерных дискретных массивов введена операция конкатенации &. Она используется при работе со строками. Тип *строка* является предопределенным типом

subtype *натуральный* **is** *целый* **range** 1.. *целый'последний*;
type *строка* **is array** (*натуральный* **range** < >) **of** *символьный*;

Пусть имеется
a : строка(1..5) := *"СЛАДКИЙ"*.
b : *строка*(1..4) := *"КИСЛЫЙ"*;
c : *строка*(1..14);

тогда

c := *a*&*"И"*& *b*;

присвоит переменной *c* значение *«СЛАДКИЙ И КИСЛЫЙ».* Хотя этот вид работы со строками ограничивается требованием, чтобы длина массива соответствовала спецификации (например»

с := *«КИСЛЫИ»* незаконно), типы без ограничений могут быть формальными параметрами процедур, и в этом случае границы индекса определяются границами фактического параметра, задаваемого в обращении. Это дает возможность пользователю программировать при необходимости свои собственные операции над строками. Кроме того, специфицируемые в ограничениях индекса границы могут быть переменными выражениями. Следовательно, Ада поддерживает динамические массивы, что также полезно в приложениях, где требуется обработка строк. Используемая для спецификации регулярных составных значений нотация является очень гибким и мощным средством. Предоставляются два основных механизма: позиционный и именной. В первом случае каждая компонента составного значения специфицируется по порядку. Во втором — каждая компонента связывается явно с индексом, например,

1 = > 8

указывает, что компоненте с индексом 1 приписывается значение 8. Нескольким компонентам может быть задано одно и то же значение, например,

3 | 5 | 10 = > 0

или, если компоненты следуют без пропусков, может использоваться обозначение для диапазона, например,

4..8=> 1

Поэтому следующие присваивания эквивалентны:

*век*1 := (0, 0, 3, 3, O, 1, 1, 1, 1, 1); *-позиционный*
*век*1 := (1 | 2 | 5 = > 0, 3 | 4=> 3, 6..10 => 1); *-именной*

Кроме того, можно использовать обозначение **others** для указания всех компонент, которые не были названы явно. В этом случае составное значение должно квалифицироваться своим подтипом, например,

*век*1 := *мойвек'*(3 | 4 = > 3, 6..10 => 1, **others** = >0);

дает в результате то же самое, что и предыдущее присваивание. Наконец, составные значения могут вкладываться в составные значения, образуя многомерные регулярные значения, например

v.матрща(1..2, 1..2) := ((0.0, 1.0), (1.0, 0.0));

Комбинированные типы описываются с помощью обычных обозначений языка Паскаль, например

type *ид_месяца* is (*янв, фее, март*,…,*дек*);
type *типдаты* **is record**
день: *целый* **range** 1..31;
месяц: *ид_месяца*;
год : *целый* **range** 1901. .2099;
end record;

Доступ к компонентам записи осуществляется с помощью обычного употребления точки, например если задано

сегодня: *типдаты*;

то

сегодня.месяц := *июнь*;

присваивает значение *июнь* компоненте *месяц* записи *сегодня.* Комбинированные составные значения формируются способом» аналогичным способу формирования регулярных составных значений, например

сегодня :== (18, *июнь,* 1981); можно использовать и именное обозначение:

сегодня := (*месяц* => *июнь, день* = > 18, *год* => 1981);

Интересной особенностью Ады является то, что для каждой компоненты можно специфицировать ее начальное значение по умолчанию. Например, имея

type *комплексный* **is record**
действ: *плавающий* := 0.0;
мн : *плавающий* := 0.0;
end record;

компилятор автоматически присвоит вещественным и мнимый частям каждого объекта типа *комплексный* значение нуль.

Комбинированный тип может быть описан с одним или более дискриминантом, обеспечивающим параметризацию типа. Параметризовать можно границы регулярных компонент и вариантные части записи. Например,

type *буфер*(*размер*: *целый* **range** 1..*макс* := 10) **is**
record
хв, хиз: *целый* **range** 1..*макс* := 1;
данные : **array** (1.. *размер*) **of** *элемент*;
использов : *целый* **range** 0..*макс* := 0;
end record;

описывает комбинированный тип, который представляет ограниченный буфер. Описание объекта, такое, как *b*1: *буфер*; создает запись, *размер* которой равен начальному значению по умолчанию 10 и которая представляет буфер, содержащий эле-

менты от 1 до 10. Дискриминантное значение может быть впоследствии изменено присваиванием всей записи при условии, что оно останется в диапазоне 1 .. *макс.* Чтобы это стало возможным, объекту *b*1 на самом деле отводится столько памяти, сколько требуется для размещения регулярной компоненты длиной *макс* элементов. Поэтому размер дискриминанта представляет скорее логическую верхнюю границу, чем физическую верхнюю границу. Когда требуется буфер постоянного размера, ограничение на дискриминант может быть задано в его описании либо непосредственно, как в примере

*b*2 : *буфер*(*размер* = > 100);

либо введением комбинированного подтипа, как в примере

subtype *буф*100 **is** *буфер*(*размер* = > 100);
*b*2:*буф*100;

Если дискриминант один раз был зафиксирован ограничением, то в дальнейшем он изменен быть не может; поэтому в последнем примере компилятору нужно отвести память только для 100 регулярных элементов, а не заботиться о размещении числа элементов *макс.*

Вариантные записи позволяют выбирать различные компоненты в зависимости от значения дискриминанта. Например,

```
type статус is (работающий, безработный);
type личность(рабстатус : статус := работающий) is
  record
     датарождения: типдаты;
     case рабстатус is
       when работающий = >
          область_налог : код_область_налог;
         код_налог : номер_код_налог;
       when безработный = >
          соц_код : соц_обеспеч_код;
     end case;
   end record;
```

описывает вариантную запись *личность.* Объекты типа *личность* можно описывать так:

*p*1 : *личность*;

при этом либо будет применяться дискриминант по умолчанию, либо может быть задано начальное значение, которое подавляет значение по умолчанию, например

*p*2 : *личность* := (*безработный,*(21, *янв,* 1946), 438291);

Как и в случае дискриминантов для массивов, на дискриминант может быть наложено ограничение, т. е.

```
subtype рабочий is личность(работающий);
```

описывает подтип *личность,* в котором дискриминант фиксирован и имеет значение *работающий.* Следует заметить, что правила языка Ада требуют, чтобы при описании комбинированного объекта дискриминант был всегда указан (см. разд. 3.3.2, где более полно обсуждаются проблемы надежности типов вариантных записей).

Наконец, Ада обеспечивает механизм динамической памяти, базирующийся на использовании ссылочных типов (указателей). Ссылочный тип обеспечивает доступ к динамически созданным объектам. Например, связанный список целых переменных можно определить следующим образом:

```
type ячейка;
type связь is access ячейка;
type ячейка is
   record
      знач: целый;
      след: связь;
   end record;
```

Обратите внимание на использование неполного описания типа для *ячейка,* позволяющее получить определение до возникновения проблемы взаимно рекурсивных определений (см. разд. 3.5.2.1). Объект типа *ячейка* создается динамически с помощью генератора, например, так:

```
первый : связь := new ячейка(0, null);
```

Явная расшифровка ссылочных значений не требуется, поэтому компонента *знач* ячейки, выбранной с помощью *первый,* обозначается так:

```
первый.знач
```

Ссылочные значения присваиваются друг другу обычным способом. Например, если *этот* описывается также ссылочным типом *связь,* то

```
этот := первый;
```

приведет к тому, что обе ссылочные переменные будут ссылаться на один и тот же объект типа *ячейка,* а

```
этот . all := первый . all;
```

присваивает значение объекта, выбранного с помощью *первый,* объекту, выбранного с помощью *этот.* В общем случае все дина-

мически выбираемые объекты перестают существовать только тогда, когда происходит выход из области действия ссылочного типа. Впрочем, Ада позволяет программисту явно управлять распределением и перераспределением памяти с помощью спецификаций представления, указаний и процедуры *непроверяемое_распределение* (см. также разд. 3.5.2.3).

13.2.4. Классическая программная структура

Операторы управления языка Ада следуют основным принципам, выдвинутым в разд. 4.2. Каждая конструкция обладает четким и однозначным синтаксисом с явной закрывающей скобкой. Оператор **if** имеет вид

```
if  условие1 then
   S1
elsjf условие2 then
   S2
   ... и т. д.
else
   Sn
end if;
```

где *Si* обозначает последовательность операторов, а части **elsif** и **else** можно употреблять или не употреблять по желанию. Оператор **case** имеет форму, аналогичную той, которая использовалась в вариантных записях, т. е.

```
case выраж is
   when выбор1 = > S1;
   when выбор2 = > S2;
      и т. д.
end case
```

где *выраж* представляет значение дискретного типа, а выбор каждого варианта зависит от одного или более значений или от диапазона значений. Например,

```
case сегодня is
   when пон..пят = > идти_на_работу;
   when суб = > мыть_авто;
   when вос = > null;
end case;
```

Утверждение **null** обозначает, что в случае *сегодня = вос* никаких действий выполнять не нужно. Каждое значение типа или подтипа в переменной **case** должно быть представлено явно в выборе варианта. Чтобы не перечислять в случае длинного списка все варианты, для которых не требуется выполнять

каких-либо действий, можно включить выбор с ключевым словом **others.** Например,

```
case i is
   when 1 = > увеличить;
   when 2 = > уменьшить;
   when others = > null;
end case;
```

Повторения обеспечиваются оператором **loop,** перед которым может стоять условие **while** или префикс **for.** Различные формы этого оператора иллюстрируются на следующих примерах:

```
loop                                    --бесконечный цикл
   читать(данные);
   обработать(данные, результаты);
   выдать(результаты);
end loop;
while этот.следующий /= null loop
   этот := этот.следующий;              --найти конец списка
end loop;
for i in 1..100 loop                    --i локализована в цикле
   c(i) := a(i) + b(i);                 --векторная сумма
end loop;
```

Кроме этого имеется оператор **exit** вида

```
exit имя_цикла when условие;
```

где как *имя-цикла,* так и условие **when** не обязательны.

Имя цикла указывается так, как показано на следующих примерах с вложенными циклами:

```
цикл : loop
          ...
          loop
             ...
             exit цикл when готово;
          end loop;
       end loop цикл;
```

Такое использование имен позволяет завершать несколько уровней циклов. Когда имя цикла опущено, завершается непосредственно обрамленный цикл.

В языке Ада есть оператор **goto** локального перехода. Причины его включения в язык не очевидны, особенно если учесть наличие других разнообразных структур управления.

Наконец, блок разрешается описывать везде, где можно описывать оператор; он может вводить новые локальные объекты

и, возможно, подпрограмму обработки исключений (см. разд. 13.2.8). Например,

```
declare
   врем: типдата;
begin
   врем := x; x := y;  y :=  врем;  --взаимно переписать x и y
end;
```

Если никаких локальных объектов не требуется, то часть **declare** можно опустить. Блок можно также пометить тем же способом, что и цикл; это позволяет ссылаться из внутренних блоков на объекты, которые стали недоступными в связи с новым использованием их имени, например,

```
внешний : declare
               x: целый;
             begin
               …
               declare
                  x: плавающий;
                begin
                  x := 1.0;                    --плавающий x
                  внешний.x := 0;          --целый x
                end;
             end внешний;
```

В языке Ада имеются два вида подпрограммных единиц: процедуры и функции. В основном они традиционны, в частности, применяются правила открытой области действия; впрочем, предлагается и несколько расширенных возможностей. Базовые формы иллюстрируются на следующих двух примерах:

```
--рассортировать массив типа вектор
   procedure сорт(x: in out вектор) is
      врем: целый;
   begin
      for i in x’первый + 1.. x’последний loop
         for j in reverse i .. x’последний loop
           if x(j - 1) > x(j ) then
              врем := x(j);
              x(j) := x(j - 1);
              x(j - 1) := врем;
           end if;
         end loop;
      end loop;
   end сорт;
```

```
--найти наибольшее значение в массиве типа вектор

 function наибольший(x : вектор) return целый is
   макс : целый := 0;
 begin
   for i in x'диапазон loop        --x'диапазон дает
     if x(i) > макс then           --диапазон фактического
       макс := x(i);               --параметра, задаваемого для
     end if;                       --в обращении  endf loop;
   end loop;
   return макс;                    --возвращает значение
 end наибольший;                   --функции
```

Классы параметров - это либо **in,** либо **out**, либо **in out** с семантикой такой, какая была определена в разд. 4.4.2; отличие состоит в том, что в соответствии со спецификациями языка простые типы передаются с помощью копирования, а механизм, который используется для других типов, зависит от реализации. Когда класс параметра не указан, считается, что неявно задан класс **in**. Параметры функций могут принадлежать только классу **in.** Ада разрешает отделять спецификацию подпрограммы от ее тела, что позволяет программисту более свободно размещать текст своей программы и обеспечивает возможность взаимно рекурсивных обращений процедур. Поэтому можно написать

```
    procedure P(x : элемент);      --спецификация
    --другие описания
    procedure P(x : элемент) is   --реализация
      --тело P
  end P;
```

В подпрограммах языка Ада необычной является спецификация фактических и формальных параметров. Во-первых, обращение к процедуре может специфицировать фактические параметры либо с помощью обычной позиционной нотации, либо с помощью именованного объединения как для составных значений. Например, если задано

```
subtype номердор is целый range 0..77;
type ид_секция is (сис, поддержка, пользов1, пользов2);
procedure поиск_диска(секция : ид.секция, нт : номердор);
```

то обращение к процедуре *поиск_диска* с номером дорожки 8 на секции *сис* можно написать любым из следующих способов:

```
   поиск_диска(сис, 8);
   поиск_диска(нт = > 8, секция = >сис);
   поиск_диска(секция = > сис, нт =>8);
```

Во-вторых, для параметров класса **in** можно специфицировать значения по умолчанию. Поэтому если приведенная выше процедура переопределена следующим образом:

procedure *поиск_диска*(*секция* : *ид_секция* := *сис*; *нт* : *номердор*);

то обращение за поиском трека 8 на секции *сис* можно написать более просто:

поиск _диска(*нт* = >8):

где параметр *секция* примет по умолчанию значение *сис.*

И последнее, подпрограммы языка Ада могут быть совмещенными (см. разд. 4.5.2). В частности, можчо совмещать предопределенные операции, иепользуя обозначение для функций. Например, предположим, что нужно расширить операцию + так, чтобы она разрешала сложение двух векторных типов. Это может быть сделано следующим образом:

```
function "+"(*, y : вектор) return вектор is
begin
  --проверить соответствие границ x и y
  declare
    результат : вектор(x'диапазон); --динамический массив
  begin
    for i in x'диапазон loop
      результат(i) :== x(i) + y(i);
    end loop;
    return результат;
  end;
end "+";
```

Заметим, что операция „=” может быть совмещенной только для типов **limited private** (см. разд. 13.2.5) и что операция „/=” совмещенной не может быть никогда, она доступна неявно как дополнение к операции „=”.

13.2.5. Абстракция данных

Конструкция языка Ада, соответствующая обсужденной в гл. 5 конструкции модуля, называется пакетом и состоит из двух отдельных частей: спецификации пакета и тела пакета. Тело может быть опущено, если не требуется никаких других деталей, кроме тех, которые приводятся в спецификации. Самым простым примером этого является пакет, используемый для представления пула данных. Для иллюстрации основного

формата пакета перепишем на языке Ада стековый модуль иа разд. 5.3.1, который теперь будет иметь вид:

```
package стек is  -часть спецификации
   procedure втолкнуть(x: in элемент);
   procedure вытолкнуть(x : out элемент);
end стек;

--другие описания

package body стек is  --часть реализации
   type сткидх is range 0..100;
   s : array(сткидх: range 1.. 100) of элемент;
   sp : сткидх := 0;
procedure втолкнуть(x : in элемент) is
begin
   sp := sp+1; s(sp) := x;
   end втолкнуть;
procedure вытолкнуть(x: out элемент) is
begin
   x := s(sp); sp := sp - 1;
end вытолкнуть;
end стек;
```

В спецификации пакета перечисляются все те объекты, которые видимы извне пакета. Все объекты внутри тела пакета являются для пакета полностью приватными. Тело пакета может содержать инициализацию, которая выполняется при обработке описания пакета. Например, приведенное выше тело пакета могло бы быть записано следующим образом:

```
package body стек is
   …
sp : сткидх;
   …
begin
   sp := 0;
end стек;
```

где стековый указатель инициализируется явно, а не косвенно в описании стека. Заметьте также, что пакет может свободно использовать объекты, описанные в содержащей пакет среде, т. е. нет списка использования, такого, как описанный в гл. 5. Чтобы показать, насколько мощна конструкция пакета языка Ада, рассмотрим, как можно расширить наш пример со стеком, сделав его более полезным. Во-первых, пакет можно обобщить, если позволить ему экспортировать тип *стек* и операции

над ними, а не сводить его действие к неявному представлению одного стекового объекта.

```
package тип_стек is
   type сткидх is range 0..100;
   type стек is
     record
       s: array (сткидх range 1..100) of элемент;
       sp : сткидх := 0;
     end record;
   procedure втолкнуть(x : in элемент; стк : in out стек);
   procedure вытолкнуть(x : out элемент; стк : in out стек);
end тип_стек;
```

Далее пользователь может описывать объекты типа *стек* и производить операции над этими объектами с помощью определенных операций *втолкнуть* и *вытолкнуть,* например,

```
declare
   use тип_стек;   --сделать непосредственно видимыми
                         объекты пакета
   s : стек              --создать стековый объект
begin
  втолкнуть(элемент1, s);
      ... и. т. д.
end;
```

Для обозначения объектов пакета обычно используют обозначения, аналогичные обозначениям записи, например, *тип_стек, втолкнуть, тип_стек.стек* и т. д. Предложение **use** позволяет опускать префикс имени пакета.

В приведенной спецификации пакета *тип_стек* представлены все подробности способа реализации типа. Обычно от этого хотят избавиться, и поэтому в языке Ада можно описывать все подробности типа в приватной части спецификации пакета. Например, *тип_стек* можно записать так:

```
package тип_стек is
  type стек is private;
  procedure втолкнуть(x : in элемент; стк : in out стек);
  procedure вытолкнуть(x : out элемент; стк : in out стек);
private
   type сткидх is range 0..100;
   type стек is
      record
        s : array (сткидх range 1.. 100) of элемент;
        sp : сткидх := 0;
      end record;
end тип_стек;
```

Во второй формулировке *тип_стек* для пользователя пакета видимо теперь только имя стека, а внутренние компоненты стекового объекта пользователю уже недоступны. Кроме операций, определенных в спецификации пакета, над приватными типами разрешаются только операции присваивания и проверки равенства и неравенства. Их можно запретить, если специфицировать тип как **limited private.**

Приватные типы можно специфицировать с помощью дискриминантов. Например, запись *тип_стек* можно параметризовать так, чтобы разрешить пользователю специфицировать требуемый размер его стека:

```
package тип_стек is
  максразмер : constant целый := верхний_ предел;
  type сткидх is range 0..максразмер;
  type стек(размер : сткидх) is private;
  procedure втолкнуть(x : in элемент; стк : in out стек);
  procedure вытолкнуть(x : out элемент ; стк : in out стек);
private
   type стек(размер : сткидх) is
     record
       s : array (1..размер) of элемент;
       sp : сткидх := 0;
     end record;
end тип_стек;
```

Теперь пользователь может описывать стековые объекты заданного размера, например,

```
use тип_стек;
s : стек (100);
большие: стек(10000);
```

И наконец, дополнительная параметризация обеспечивается в языке Ада с помощью возможностей определения настройки. Настраиваемый пакет (допускаются также и настраиваемые подпрограммы) это шаблон для генерации фактических пакетов, в которых используемые в шаблоне формальные параметры настройки замещаются фактическими значениями (см. разд. 9.6.4). Рассмотрим снова наш пакет *тип_стек*; он может обрабатывать объекты только типа *элемент.* Если *элемент* сделать параметром настройки, то появляется возможность создавать стек для любого типа данных.

```
generic
    type элемент is private;   --параметр настройки
package тип_стек is
   --как прежде
end тип_стек;
```

Например, пользователь может создать стек для 100 символов.

оформляя сначала пакет *тип_стек* для символов и затем описывая соответствующий стековый объект:

```
package тип_стек_сим is new тип_стек(символьныи);
                          --создать фактический пакет
use тип_стек_сим:         --открыть его
  cs : стек(100);         --использовать его
```

Настраиваемые пакеты и подпрограммы позволяют параметризовать во время компиляции значения, имена, типы и подпрограммы. Они удобны, например, при создании библиотек, особенно пакетов ввода-вывода.

13.2.6. Параллельность

Язык Ада позволяет вводить в программу параллельные задачи (так в языке называется процесс), используемые при этом обозначения аналогичны обозначениям лля$_x$ пакетов. Связь между задачами осуществляется с помощью расширенного рандеву. Запросы на передачу данных принимаются в вызываемых задачах операторами приема **accept.** В Аде они называются входами, каждый вход должен быть перечислен в спецификационной части задачи. Недетерминизм реализуется с помощью оператора выбора **select,** перед каждой альтернативой выбора может стоять охрана. Основные понятия механизма рандеву были объяснены в разд. 6.5. На нескольких примерах проиллюстрируем используемую в языке Ада нотацию.

Следующая задача обеспечивает взаимно исключающий, доступ к общей переменной:

```
task защищенная_пер is                --спецификация
     entry читать(x : out данные);
     entry писать(x : in данные);
end защищенная_пер;

task body защищенная_пер is            --тело
    общая_пер: данные;
begin
  loop
    select
      accept читать(x: out данные) do
          x := общая_пер;
      end читать;
    or
      accept писать(x : in данные) do
         общая_пер := x;
      end писать;
    end select;
   end loop;
 end эащищенная_пер;
```

В спецификации задачи перечисляются входы, которые могут быть вызваны для данной задачи (порядок объектов в спецификации задачи не предусматривается). Тело задачи состоит из бесконечного цикла, принимающего обращения к *читать* и *писать* в произвольной последовательности. В задачу может быть включено более одного оператора **accept,** соответствующего одному входу; поэтому тело рассматриваемой задачи можно переписать следующим образом:

```
task body защищенная _пер is
   общая_пер : данные;
begin
   accept писать(x : in данные) do
      общая_пер := x;
   end писать;
   loop
      --операторы цикла, как выше
   end loop;
 end защищенная_пер;
```

этим обеспечивается то, что первое обращение, принятое процедурой *защищенная-пер,* запишет в общую переменную начальное значение.

Приведенное описание задачи может быть включено в описательную часть любой подпрограммы на языке Ада, в любой пакет, любую задачу или блок. В результате этого включения непосредственно вводится и инициализируется заданный объект задачи (не требуется явного применения оператора инициализации). Например, рассмотрим следующий эскиз блока:

```
вилка: declare
          task защищенная-пер is ...
          task body защищенная_пер is ...
          --возможно другие описания
          --включения других задач
       begin
          --
          --
          защищенная_пер.писать(...);   -обращение
                                         --к входу задачи
       end вилка;
```

Непосредственно перед выполнением первого оператора в теле процедуры *вилка* будет активирована задача *защищенная_пер.* Обращения к ее входным процедурам могут делаться как из тела процедуры *вилка* (это показано в примере), так и из лю-

бой другой задачи, описанной внутри блока *вилка.* При достижении конца блока выполнение задачи, в которой была описана *вилка,* приостанавливается. В дальнейшем она может покинуть блок только при завершении всех содержащих ее задач. Как правило, задача завершается, когда она доходит до своего конца; но обслуживающие задачи, такие, как *защищенная_пер,* традиционно записываются как бесконечные циклы. Следовательно, в нашем примере управление никогда не выйдет из блока. Эта проблема решается в языке Ада с помощью введения специальной альтернативы **terminate,** которая включается в оператор **select,** например:

```
task body защищенная_пер is
    общая_пер: данные;
begin
   accept писать(x : in данные) do
     общая_пер := x;
   end писать;
   loop
     select
       accept читать(x: out данные) do
           x := общая_пер;
       end читать;
     or
       accept писать(x : in данные) do
          общая_пер := x;
       end писать;
     or
       terminate
     end select;
    end loop;
 end эащищенная_пер;
```

Каждый раз при выполнении оператора **select** либо будет приниматься обращение к *читать* и *писать,* либо, если (и только если) достигнут конец содержащего блока, задача будет завершена.

Как видно из приведенного выше примера, базовая синтаксическая конструкция обращения к входу аналогична обращению к процедуре, где перед именем входа пишется имя задачи, например,

Защищенная_пер.писать(*x*)

Предложение use не может быть использовано для обеспечения непосредственной видимости имен входов задачи, однако вход может быть переименован как процедура следующим образом:

```
procedure обновить(x; in данные) renames защищенная_пер.
                                                  писать;
```

В связи с этим требуемый эффект можно получить, написав

обновить(*x*)

Пользовательской задаче может оказаться необходимым сделать обращение к входу только в том случае, когда имеется гарантия, что это обращение будет принято немедленно. Ада разрешает делать такие условные обращения к входу с использованием обозначений оператора выбора, например,

```
select
   обновить(x)
else
   --выполнить нечто другое
end select;
```

Кроме этого, можно сделать обращение к входу, накладывая временные ограничения:

```
select
   обновить(x)
or
   delay 1.0;
   --выполнить нечто другое
end выбрать;
```

В этом случае обращение к процедуре *обновить* будет происходить только в том случае, если оно может быть принято в течение специфицируемого периода задержки, в противном случае выполняются операторы, следующие за оператором задержки. Оператор задержки можно также использовать аналогичным способом внутри обслуживающих задач для выполнения альтернативных действий в случае, когда в течение заданного периода задержки не происходит никаких обращений к входам. И наконец, Ада позволяет описывать задачу не как объект, но скорее как тип. Этим обеспечивается возможность размножения задач. Типы задач могут использоваться в записях и массивах, вместе с типами доступа они обеспечивают возможность динамического создания задач. В качестве примера рассмотрим использование типа задачи для реализации семафора. Он может быть определен следующим образом:

```
task type сем is
   entry оградить;
   entry освободить;
end сем;

task body сем is
   занят : логический := ложь;
begin
  loop
    select
        when not занят = >    --охраняемая альтернатива
          accept оградить do
            занят := истина;
          end;
    or                            --эта альтернатива без охраны
      accept освободить do
        занят := ложь
      end;
    or
       when not занят = >      --охраняемая альтернатива
          terminate;
    end выбрать;
  end loop;
end сем;
```

В дальнейшем можно строить защищенные переменные данных, включая данный тип задачи в запись в качестве компоненты, например,

```
type ресурс is
record
   s : сем;  d : данные;
end record;
```

Каждое последующее описание объекта типа *ресурс* вызывает создание экземпляра *сем* типа задачи, например,

```
declare
   rl, r2, r3: ресурс;
begin
   --теперь активны 3 экземпляра задачи сем
   r1.s.оградить;            --оградить r1
   обработать(r1.d);   --обработать его данные
   r1.s.освободить;          --освободить rl  -и т. д.
end;
```

13.2.7. Программирование ввода-вывода низкого уровня

В языке Ада предоставляется способ отображения перечислимых типов и комбинированных типов на физическую память, при этом имеется возможность управления с помощью спецификации представления. В случае перечислимых типов представление специфицирует тот внутренний код, который должен быть использован. Например, для перечисления

```
type классrl is (замо_ctr, r1_выс, r1_низ, r1_оба);
```

(см. разд. 7.2.2) внутренние значения для каждой перечислимой константы можно специфицировать, написав

```
for классrl use (замок_ctr = > 0, rl_выс = >1,
                 rl_низ = > 2, rl_оба => 3);
```

В случае записей место каждой компоненты специфицируется в терминах сдвигов от начала записи, измеряемых в единицах памяти и диапазона положений битов в данном кванте памяти. Поэтому для типа *класс_таймер,* определенного (см. разд. 7.2.2) так:

```
type класс_таймер is
  record
    elm : класс_счетчик;
    fnc : функция_таймер;
    rlm : классrl;
    ctr : целый range 0..2;
  end record;
```

его отображение можно специфицировать следующим образом:

```
for класс_таймер use
  record
    ctm at 0 range 0 . . 0;
    fnc at 0 range 1 . . 3;
    rlm at 0 range 4 . . 5;
    ctr at 0 range 6. .7;
  end record;
```

Кроме этого, спецификация записи может содержать предложение выравнивания; в результате его выполнения каждая запись заданного типа выравнивается по начальному адресу, значение которого кратно значению, заданному в предложении выравнивания.

Адрес, по которому должен быть помещен объект в памяти, можно специфицировать следующим образом:

```
регбуф : символьный;
for регбуф use at 8#177566#;
```

что располагает переменную *регбуф* по восьмеричному адресу 177566. Спецификации адреса могут также применяться к обращениям входов внутри задач для связи с адресом вектора прерывания, как, например,

```
task часы is
  entry часы_прер;
  for часы_прер use at 8#100#;
end часы;
```

и для возможности расположения системных процедур (написанных не на языке Ада) по фиксированным адресам памяти, как, например,

```
procedure подпрограмма_ядра;
for подпрограмма_ядра use at 16#20#;
```

Кроме перечисленных выше, предоставляется еще ряд возможностей несколько более специального характера, таких, как вставки кода, указания и системно определенные подпрограммы ввода-вывода низкого уровня. Можно отметить также, что допускаются спецификации представления для управления распределением памяти, используемые, в частности, при распределении памяти для набора динамических объектов данных. Другие примеры использования возможностей программирования низкого уровня языка Ада можно найти в примере программы, приведенной в разд. 13.3.

13.2.8. Обработка исключений

Язык Ада предоставляет механизм обработки исключений, аналогичный механизму, описанному в гл. 8. Подпрограммы обработки исключений могут содержать любой блок, любая подпрограмма, любое тело пакета или задачи. Используемые при этом обозначения иллюстрируются на следующем эскизе^

```
procedure исключения is
   e1, e2, e3 : exception;
   …
begin
   …
exception
   when e1 = >
      --действия для ошибки e1
   when  e2 | e3 = >
      --действия для ошибок e2 и e3
when others = >
      --действия для любых других ошибок
end исключения;
```

Исключения могут определяться пользователем (как в только что приведенном примере) или быть предопределенными. Примеры второй возможности следующие:

```
ошибка_ограничения -- нарушение ограничений диапазона,
                   --индекса или дискриминанта
числовая_ошибка    --переполнение, „недополнение" и т. д.
ошибка_выбора      --закрыты все альтернативы в операторе
                   --выбора
ошибка_памяти      -- исчерпание памяти при динамическом
                   --распределении
ошибка_задач       --ошибка связи между задачами
```

Определенные пользователем исключения вызываются с помощью оператора **raise:**

```
raise e1;
raise;  -- вновь вызвать последнее вызванное исключение
```

Последняя форма может появиться только в подпрограмме обработки исключений. Предопределенные исключения могут возбуждаться либо в декларативной части блока, либо внутри основного тела. В первом случае обработка описательной части прекращается и исключение возбуждается в объемлющем блоке. Во втором (обычном) случае исключение возбуждается внутри блока и управление передается подпрограмме обработки исключения, приводимой в конце блока, если таковая имеется. Поиск подпрограммы обработки исключения осуществляется динамически в точности в соответствии с правилами, приведенными в разд. 8.3.3. Соответственно, если в текущем блоке не имеется подпрограммы обработки данного конкретного исключения, блок завершается и исключение перевозбуждается в вызывающем блоке. Этот процесс продолжается до тех пор, пока не находится подпрограмма обработки исключения или не достигается самый внешний уровень. Если на самом внешнем уровне имеется библиотечная единица (т. е. пакет или подпрограмма), то выполнение основной программы прекращается. Если это тело задачи, то задача завершается и исключение дальше не распространяется.

В качестве примера использования исключений рассмотрим следующую пересмотренную версию пакета *тип_стек* из разд. 13.2.5, в котором определены исключения *стек_полон* и *стек_пуст,* позволяющие пользователю пакета обрабатывать любые появляющиеся ошибки.

```
package тип_стек is
  type стек is private;
  procedure втолкнуть(x : in элемент; стк : in out стек);
  procedure вытолкнуть(x : out элемент: стк : in out стек);
 стек_полон, стек_пуст : exception;
private
   type сткидх is range 0..100;
   type стек is
     record
        s : array (сткидх range 1..100) of элемент;
        sp : сткидх := 0;
      end record;
```

```
end тип_стек;

package body тип_стек is

  procedure втолкнуть(x : in элемент; стк : in out стек) is
  begin
    if стк.sp= 100 then
      raise стек_полон;
    end if;
    стк.sp:= стк.sp + 1;
    стк.s(стк.sp) := x;
  end втолкнуть;

  procedure вытолкнуть(x : out элемент; стк: in out стек) is
  begin
    if стк.sp = 0 then
      raise стек_пуст;
    end if;
     x := стк.s(стк.sp);
    стк.sp:= стк.sp - 1;
   end вытолкнуть;

end тип_стек;
```

Пользователь стекового пакета задаст подпрограммы обработки для исключений *стек_полон* и *стек_пуст,* например, следующим образом:

```
procedure взять_стек;
  use тип_стек;
  s : стек;
  x : элемент;
begin
  втолкнуть(x, s);
  вытолкнуть(x, s);
  ... и т. д.
 exeption
   when стек_полон = >
      занести («Переполнение стека!»);
  when стек_пуст = >
      занести(«Стек пуст!»);
 end взять_стек;
```

В задачах исключения обрабатываются обычным способом, кроме того случая, когда они встречаются при связи задач. Если исключение возбуждается в операторе приема и не обрабатывается локально оператором приема, то оно передается вне в объемлющий блок и также к вызывающей задаче.

Задача *T* может быть завершена не обычным способом некоторой другой задачей, возбуждающей предопределенное исключение *T' нарушение.* Это единственное исключение, которое может быть возбуждено в задаче непосредственно некоторой

другой задачей. Если нарушившая задача обратилась ко входу в некоторой другой задаче, то в случае, когда вызов еще не был принят, обращение просто отменяется; в противном случае исключение задерживается до тех пор, пока не завершится рандеву. Если же, с другой стороны, нарушившая задача выполняет оператор приема и не имеется никакой локальной подпрограммы обработки для этого исключения, то рандеву прекращается и возбуждается исключение *ошибка_задачи* в вызывающей задаче. Можно отметить, что возбуждение *T' нарушение* не гарантирует, что принимающая задача завершится, — она может просто обработать исключение и продолжать дальше. Эта ответственность за завершение возлагается на нарушившую задачу, чтобы завершиться чисто, она может выполнить свое «последнее желание». Если задаче не удается ответить на исключение нарушения, ее завершение может быть вызвано с помощью оператора **abort** *T*;. Впрочем, это крайняя мера, к которой следует обращаться в исключительных случаях.

13.2.9. Раздельная компиляция

Язык Ада предоставляет набор возможностей для раздельной компиляции, что позволяет строить большие программы либо методом сверху вниз, либо снизу вверх, либо и тем и другим методом одновременно. Этот механизм разработан так, что использование раздельно откомпилированных модулей не влияет на надежность системы в целом. Компилятор поддерживает базу данных спецификаций модульного интерфейса и вспомогательной среды, так что может быть выполнена полная проверка интерфейса.

Компилируемые компоненты бывают двух сортов: библиотечные компоненты и вспомогательные компоненты. Библиотечная компонента — это компонента, которая описывается на самом внешнем уровне программы. Вспомогательная компонента — это отдельно компилируемое тело подпрограммы, пакета или задачи, описанное на самом внешнем уровне другой компилируемой компоненты (либо библиотечной, либо вспомогательной).

Перед компилируемой компонентой может находиться предложение **with,** в котором указываются все компоненты, от которых зависит текущая компонента (если такие имеются). Работу этого механизма раздельной компиляции лучше всего показать на примере. Рассмотрим следующий эскиз программы, базирующийся на примере пакета *стек*:

```
procedure взять_стек is

  package стек is
     procedure втолкнуть(x: in целый);
     procedure вытолкнуть(x : out целый);
  end стек;

  package body стек is
     --
     --определить стек и процедуры втолкнуть и вытолкнуть
  end стек;

  use стек;  --открыть пакет стек
begin
  втолкнуть(1);
  втолкнуть(2);
   ... и т, д.
 end вэять_стек;
```

Эту программу можно построить методом снизу вверх в виде трех отдельных компилируемых компонент:

```
--компонента 1 (предложения with не нужно, так как нет
   зависимости от других компонент)
package стек is
   procedure втолкнуть(x: in целый);
   procedure вытолкнуть(x : out целый);
end стек;

--компонента 2 (для тела пакета предложения with не нужно—
   предполагается тот же самый контекст,  что и  для
   спецификационной части)
package body стек is
   --определить стек и втолкнуть и вытолкнуть
end стек;

--компонента 3
   with стек;  -контекст есть пакет стек
   procedure взять_стек is
      use стек;
   begin
      втолкнуть(1);  втолкнуть(2);
      ... и т. д.
   end взять_стек;
```

В данном примере пакет *стек* был разработан (и предположительно отлажен) до того, как была разработана компонента, которая его использует. Разработка программы методом сверху вниз поддерживается также с помощью вспомогательных компонент. Возвращаясь снова к примеру *стек,* предположим, что процедуры *втолкнуть* и *вытолкнуть* должны быть разработаны

позднее. Тогда тело пакета *стек* может быть записано следующим образом:

```
package body стек is
   type сткидх is range 0..100;
   s : array(сткидх range 1..100) of целый;
   sp : сткидх := 0;
   procedure втолкнуть(x : in целый) is separate;      --заглушка
   procedure вытолкнуть(x : out целый) is separate; --заглушка
end стек;
```

В дальнейшем каждая из процедур *втолкнуть* и *вытолкнуть* может компилироваться раздельно. Например, процедура *втолкнуть* могла бы быть записана так:

```
separate (стек)
procedure втолкнуть(x : out целый) is
begin
   x := s(sp); sp := sp - 1;
end втолкнуть;
```

В префиксе **separate** (раздельно) указывается родительская компонента для отдельно компилируемой подкомпоненты. Его значение заключается в создании объемлющего контекста, который идентичен контексту на месте соответствующего остатка тела. Поэтому процедура *втолкнуть* по-прежнему имеет доступ к именам *s* и *sp*. Любая подкомпонента может содержать остатки тела для других подкомпонент, следовательно, программа может строиться сверху вниз таким образом на любую глубину.

13.2.10. Ввод-вывод

В языке Ада не предоставляется никаких встроенных возможностей ввода-вывода. Однако определяется набор стандартных пакетов для выполнения широкого спектра операций ввода-вывода. Из-за недостатка места мы не можем подробно на этом останавливаться. Впрочем, читатель теперь может сам оценить, что разнообразие возможностей, предоставляемых настройками, пакетами и совмещенными подпрограммами, делает возможным определить весь ввод-вывод высокого уровня почти полностью на самом языке Ада. В результате, используя нестандартные соглашения о вводе-выводе, можно все же построить относительно мобильную программу. Все, что требуется, — это перенести вместе с программой сам пакет вводавывода.

Этим мы завершаем описание языка программирования Ада. В следующем разделе приводится достаточно обширный пример его использования.

13.3. ИСПОЛЬЗОВАНИЕ ЯЗЫКА АДА

13.3.1. Удаленная система приема данных

Чтобы проиллюстрировать использование языка Ада, опишем программу, реализующую на микрокомпьютере LSI 11 удаленную управляемую систему приема данных. Но вначале нужно обратить внимание читателя на то, что в момент написания этой книги никакой вычислительной машины, на которой реализован язык Ада, не существует. Поэтому приводимый далее пример и не испробован, и не отлажен. Хотя он и разрабатывался очень внимательно, ошибки неизбежны. Впрочем, главной целью включения в эту главу примера реальной программы является иллюстрация того, как используется язык Ада; простой пример работающей программы нам не интересен. Мы надеемся, что в этом отношении приводимый пример соответствует поставленной задаче.

Структура программного обеспечения, которое мы собираемся описывать, изображена схематически на рис. 13.1, где каждый кружочек представляет задачу в смысле языка Ада. Вся система разбита на две главные подсистемы: подсистема связи, которая обеспечивает всю связь с главной машиной, и подсистема сканирования, которая выполняет функции фактического приема данных.

Все данные, передаваемые между главной машиной и системой приема, имеют форму пакетов, где каждый пакет состоит из потока символов ASCII. Пакеты данных передаются от системы приема главной машине, они содержат фактически зарегистрированные измерения. Структура этих пакетов приведена на рис. 13.2. Система имеет не больше 64 измерительных каналов, время замеров для каждого канала конкретно определяется главной машиной. Поэтому каждый раз, когда производятся измерения, число фактических операций считывания будет меняться. Отсюда следует, что возвращаемые главной машине пакеты данных имеют переменную длину. Эта особенность была введена специально, чтобы показать возможности языка Ада при работе с динамическими массивами.

Получаемые от главной машины пакеты управления содержат одну из четырех команд: открыть канал, закрыть канал, изменить время замеров для канала, установить системные часы. Формат этих команд приведен на рис. 13.3. Все пакеты данных должны быть удостоверены сигналами АСК или NAK. Если приходит ответ NAK или не приходит никакого ответа, то пакет передается заново.

Кратко работу удаленной управляемой единицы приема данных можно сформулировать следующим образом. В выключенном состоянии все каналы закрыты и никаких измерений не

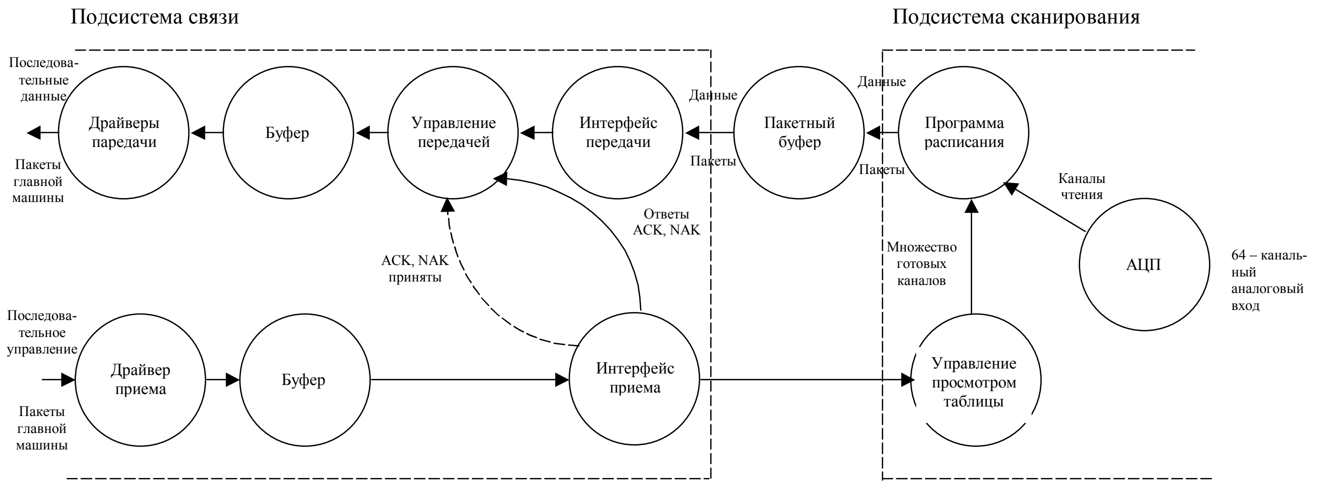


Рис. 13.1. Заданная структура системы приема данных.

производится. Затем главная машина посылает пакет управления для инициализации системных часов на истинное время. После этого она посылает команды открытия каналов, активи-

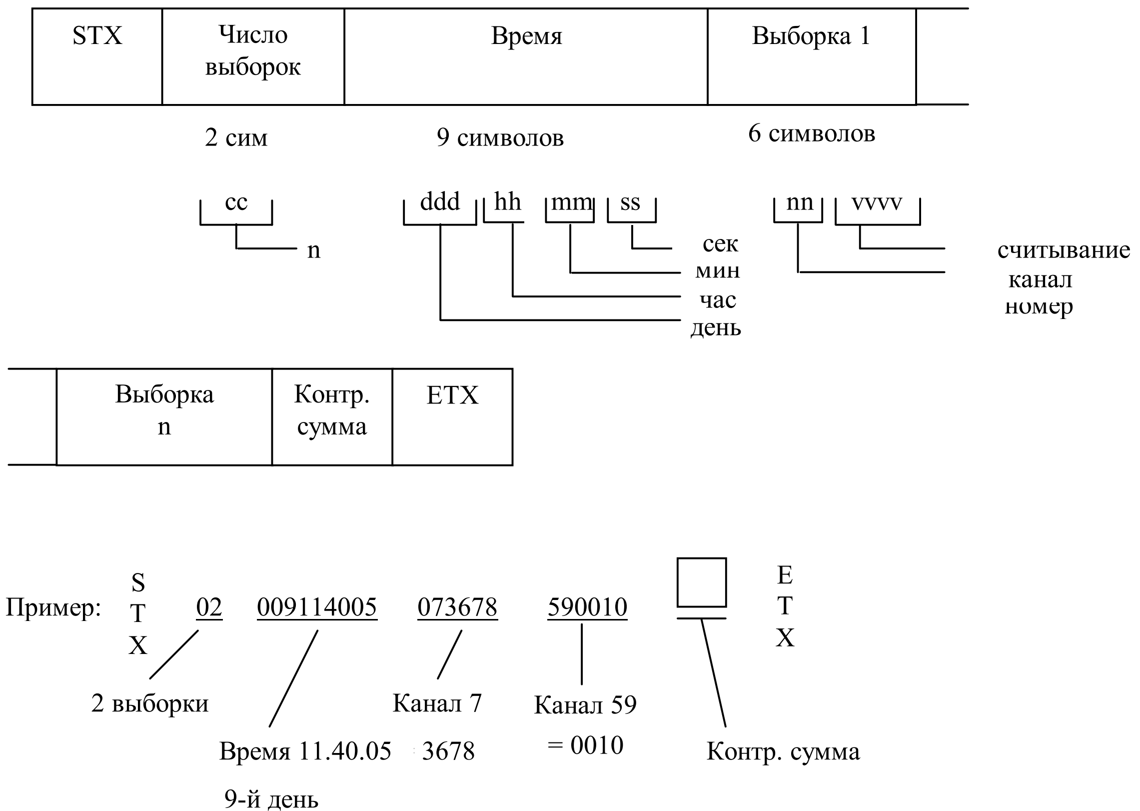


Рис. 13.2. Формат передачи пакета данных.

зирующие каждый требуемый измерительный канал и устанавливающие соответствующий период взятия замеров (как правило, от 5 до 30 секунд). Начиная с этого момента задача со-

Рис. 13.3. Формат передачи пакета управления.

ставления расписания периодически исследует каждый канал и принимает результаты для всех тех каналов, для которых настало

STX	CMD	АРГУМЕНТЫ	Контр. сумма	ETX

Открыть: O nn sss — nn = номер канала

Изменить: A nn sss — ss = период выборки

Закрыть: C nn

Установить время: T ddd hh mm ss

время производить замеры. Это множество результатов собирается затем в пакет данных, к пакету добавляется указание времени, когда производились замеры, и пакет передается

обратно главной машине. В любое время главная машина может закрыть канал, открыть новый канал или изменить период канала, послав новый пакет управления.

13.3.2. Спецификация

Новой особенностью языка Ада является способ раздельного задания спецификаций пакетов, подпрограмм и задач отдельно от их тел. Эта возможность позволяет полностью специфицировать структуру системы до решения любых мелких деталей реализации. В этом разделе описывается множество пакетов, которые совместно полностью специфицируют программное обеспечение системы приема данных. Каждый раздел кода представляет отдельную компилируемую компоненту.

Прежде чем представить фактические пакеты системы, следует заметить, что система прогона задач, написанных на языке Ада, требует специальных часов реального времени. Поэтому в отличие от предыдущего примера на языке Модула доступ к системным часам непосредственно в языке Ада не программируется, он осуществляется с помощью определенных реализацией процедур. В нашем примере предполагается наличие следующего пакета работы со временем. Он представляет тип данных с именем *время* и некоторые операции над объектами типа *время* наряду с процедурами доступа к системным часам.

```
package времясис is
   package тип_время is
      type время is
         record
            день : целый range 0..999;
            час : целый range 0. .23;
            мин : целый range 0. .59;
            сек : целый range 0. .59;
         end record;
      subtype период is целый range 0..целый'последний;
      function "+"(t: время; d : период) return время;
      function "+"(d : период; t : время) return время;
      function ">"(t1, t2 : время) return логический;
      function отоб_время{(t : время) return строка;
      function отоб_время(s : строка) return время;
      ошиб_время : exception;
   end тип_время;
   package систем_часы is
     use тип_время;
     procedure время_есть(t: in время);
     function время_теперь return время;
   end систем_часы;
end времясис;
```

Имея этот предопределенный пакет, можно начать разработку самой системы. Во-первых, в процессе разработки всей системы будет использоваться ряд констант и часто употребляемых определений подтипов. Эти элементы собираются в пакет с именем *системно_глобальный,* доступ к нему возможен из любого другого пакета системы при указании имени *системно_глобальный* в спецификации контекста:

```
package системно_глобальный is
   чис_каналов : constant := 64;
   subtype номер_канала is целый range 0..чис_каналов - 1;
   subtype счит_канала is целый range 0..4095;
   subtype позцел is целый range 0.. целый'послед;
   глав_связь_перерыв : constant продолжит := 10.0;
   распис_цикл_время : constant продолжит := 1.0;
end системно_глобальный;
```

Как было описано выше, все данные передаются между главной машиной и системой в форме последовательных пакетов. Хотя формат этих пакетов и удобен для передачи по звену последовательной связи, он не удобен для работы внутри самой программы. Поэтому введем внутреннее представление этих двух пакетных типов в форме комбинированных типов данных. Каждый тип определяется пакетом и функцией преобразования, осуществляющей отображение между внешним и внутренним форматами. Как было замечено ранее, размер пакетов данных может меняться, поэтому будем использовать комбинированный тип с дискриминантом, соответствующий динамическому массиву.

```
with системно_глобальный,  времясис;
use системно_глобальный, времясис;
package тип_пакет_данных is
   type данное is
     record
        cn: но мер_канала;
        cr : счит_канала;
     end record;
   use тип_время;
   макс : constant := чис_каналов;
   type пакет_данных(размер : целый range 1..макс := макс) is
     record
        t: время;
        данные : array (1.. размер) of данное;
     end record;
   function преобразовать(dp : пакет_данных) return строка;
end тип_пакет_данных;
```

Заметьте, что в данном фрагменте дискриминанту *размер* присваивается начальное значение. Это позволяет при необходимости описывать неограниченные объекты типа *пакет_данных* (*см.* разд. 13.2.3).

Пакеты управления представляются вариантной записью, чтобы имелась возможность отличить три команды каналов от команды *установить-время.*

```
with системно_глобальный, времясис;
use системно_глобальный, времясис;
package тип_пакет_управления is
use тип_время;
   type команда is (открыть_канал, изменить_период,
                           закрыть_канал, установить_время);
   type пакет_управления(кмд : команда := открыть_канал) is
      record
         case кмд is
            when установить_время = >
               t: время;
            when othes = >
               еп: номер_канала;
               р : период;   --0 когда кмд = закрыть_канал
          end case;
      end record;
   function преобразовать(s : строка) return пакет_управления;
   ср_формат_ош: exception;
end тип_пакет_управления;
```

Структура этого пакета аналогична структуре пакетов данных с той поправкой, что первый включает также описание исключения *ср_формат_ош,* которое может быть возбуждено во время обращения к функции *преобразовать.* В данном случае это необходимо потому, что функция *преобразовать* работает с получаемыми от главной машины пакетами, которые могут оказаться испорченными; в предыдущем случае эта функция работала с пакетами данных, которые создавались внутри системы, где возможность генерирования неправильного пакета минимальна.

Определив необходимые для нас главные типы данных, можно написать спецификации двух основных подсистем. Каждая подсистема представляется пакетом, тело которого содержит спецификации каждой из составляющих пакет задач. Тело каждой из этих задач специфицируется ключевым словом

separate, поэтому подробности их реализации могут быть определены позднее.

Подсистема связи специфицируется следующим образом:

```
--Подсистема интерфейса связи с главной машиной
package подсистема_связи is
--никаких объектов не экспортируется
end подсистема_связи;

package body подсистема_связи is
    type класс_освободить is (немедленно, после_удостов);
    --Аппаратные адреса
    tx_век : constant :=  8#64#;
    rx_век : constant := 8#60#;
    tx_ста : constant := 8 #177564#;
    tx_буф : constant := 8#177566#;
    rx_ста : constant := 8#177560#;
    rx_буф : constant := 8#177562#;
    ввести_нач : constant := 8#100#;
    снять_нач : constant :=  8#0#;
    --Драйверы устройств
    task tx_драйвер is
       entry прерывание;
       for прерывание use at  tx_век;
       pragma приоритет(4);  --предполагается, что здесь
                             --устанавливается приоритет
                             --процессора PDP11
    end tx_драйвер;
     task rx_драйвер is
        entry прерывание;
        for прерывание use at  rx_век;
        указание
        pragma приоритет(4);  --предполагается, что здесь
                              --станавливается приоритет
                              --процессора PDP11
     end rx_драйвер;
     --Буферы
     generic
        размер : in естественный;
     package сим_буфер is
         task буферизация is
            entry занести(см : in символьный);
            entry выбрать(см ; out символьный);
         end буферизация;
      end сим_буфер;

   package body сим_буфер is separate;
   package выв_сим_буфер is new сим_буфер(256);
```

```
package вв_сим_буфер is new сим_буфер(64);
procedure tx_занести(см : in символьный) renames
   выв_сум_буфер.буферизация.занести;
 procedure tx_выбрать(см : out символьный) renames
   выв_сим_буфер.буферизация.выбрать;
 procedure rx_занести(см : in символьный) renames
   вв_сим_буфер.буферизация.занести;
 procedure rx_выбрать(см : out символьный) renames
   вв_сим_буфер.буферизация.выбрать;
 --tx управление
 --Программа расписания запрашивает от задачи
 --tx_интерфейса
--передачу пакетов данных
--и запрашивает от rx_интерфейса передачу ack/nak
--ответов
task tx управление is
   entry оградить;
   entry tx(см : in символьный);
   entry освободить(осв_класс : in освободить_класс; ок : out
         логический);
   entry ответ_получен(подтвержден : in  логический);
end tx_управление;
--Задачи интерфейса
task tx_интерфейс;    --Выбирает пакеты данных из пакетного
                      --буфера и передает их.
task rx_интерфейс;    --Получает пакеты управления от главной
                      --машины и посылает их к программе
                      --управления сканирования таблицы.
                      --Получает подтверждения от главной
                      --машины и передает их управлению tx
task body tx_драйвер is separate;
task body rx_драйвер is separate;
task body tx_управление is separate;
task body tx_интерфейс is separate;
task body rx.интерфейс is separate;
end подсистема_связи;
```

Прежде всего отметим, что этот пакет никак не обслуживает другие пакеты. Другими словами, вся связь между этим пакетом и остальной частью системы осуществляется в форме процедур выхода и обращений к входам. Тело этого пакета содержит спецификации для семи задач, которые реализуют подсистему связи. Драйверы *tx_* и *rx_* являются стандартными задачами обработки прерываний, которые выполняют бесконечный цикл, обслуживая символьные буферы, с которыми они связаны. Второй из них имеет форму ограниченного циклического буфера, такого, как буфер, описанный в гл, 7, Они оба

идентичны, различны только их фактические размеры. Поэтому определяется настраиваемый пакет с именем *сим_буфер,* обслуживающий одну буферную задачу; размер этого буфера является параметром. В дальнейшем каждый фактический буфер создается как экземпляр этого пакета с двумя различными значениями переменной *размер.* Далее для удобства работы входы фактических буферов переименовываются как процедуры, *tx_занести* писать легче, чем *выв_сим_буфер.буферизация.занести*!

Вводится задача с именем *tx_управление,* которая осуществляет выбор между конкурирующими запросами от задачи *tx_интерфейс* на передачу пакета данных и от задачи *rx-интерфейс* на передачу подтверждения (т. е. АСК или NAK). Для того чтобы передать один или более символов, задача пользователя должна прежде всего обратиться к входу *оградить.* Затем она может обращаться к *tx* сколько угодно раз для передачи потока символов. При завершении передачи она должна обратиться к *освободить.* Задаваемый параметр этого входа специфицирует класс освобождения либо немедленно, либо после получения подтверждения. Логический параметр о/с указывает, была ли передача успешной.

Задача *tx_интерфейс* просто выбирает пакеты данных из буферного пакета и передает их в последовательной форме через задачу *tx_управление.* Задача *rx_интерфейс* выполняет две работы. Ее главное назначение — получать пакеты управления в последовательной форме, преобразовывать их во внутреннее представление и затем передавать их подсистеме сканирования. Кроме этого, она должна выбирать любые символы АСК или NAK, получаемые от главной машины, и передавать их задаче *tx_упраеление,* обращаясь ко входу *ответ_получен.*

Код подсистемы сканирования имеет следующий вид:

```
--Подсистема сканирования
with системно_глобальный, тип_пакет_управления, времясис;
use системно_глобалъный, тип_пакет_управления, времясис;
package подсистема_скан is
   procedure обновить(ср : in пакет_управления);
end подсистема_скан;
package body подсистема_скан is
   --Аппаратные адреса ацп
   ацп_век : constant := 8#340#;
   ацп_ста : constant := 8#170400#;
   ацп_буф : constant := 8#170402#;
   use тип.время;
   type уст_готово is array(номер_канала) of логический;
   --Программа управления таблицей сканирования
```

```
   task управление_таблицей_скан is
      entry обновить(ср : in пакет_управления);
      entry проверить(общий : out позцел; готово : out уст_готово;
                                                         t: out время);
   end управление_таблицей_скан;
   --Драйвер ацп
   task ацп is
      entry читать(n : in номер_канала; r : out чтение_канала);
      entry преобразование_завершен;
      for преобразование_завершен use at ацп_век;
      pragma приоритет(5);
    end;
   --программа расписания
   --проверяет таблицу сканирования по
   --распис_цикл_время.
   task прог_расписание;
   --привести тела в другом месте
   task body управление_таблицей_скан is separate;
   task body ацп is separate;
   task body прог_расписание is separate;
   procedure обновить(ср : in пакет_управления) is
   begin
      управление_таблицей_скан.обновить(ср);
   end обновить;
end подсистема „сканирования;
```

Как можно видеть, подсистема сканирования значительно проще подсистемы связи. Она описывается точно так же, как и подсистема связи пакетом, тело которого состоит из спецификаций составляющих задач. Необходимы всего лишь три задачи. Задача *управление_таблицей_скан* обеспечивает взаимно исключающий доступ к центральной таблице информации, а которой регистрируются статус и период выборки для каждого измерительного канала. К входу обновления обращается (косвенно) задача *rх_интерфейс* каждый раз при получении нового пакета управления. К входу *проверить* обращается задача расписания периодически для определения того, из какого канала можно выбирать измерения. Задача АЦП — это подпрограмма обработки прерывания и драйвер для аналогоцифрового преобразователя. К входу *читать* обращается программа расписания для измерения уровня входа на специфицируемом канале. Сама программа расписания совершает циклы с фиксированным периодом, определяемым (приблизительно) константой *распис_цикл_время,* описываемой в пакете *системно_глобальный.* В каждом цикле эта задача обращается к *управленив_таблицей_скан.проверить,* чтобы определить, для какого канала наступило (если наступило время чтения. Потом она

обращается к *ацп.читать* для взятия замеров с каждого канала и затем конструирует пакет данных по результатам считывания и посылает его в пакетный буфер.

И наконец, пакетный буфер специфицируется пакетом, содержащим одну задачу с входами *занести* и *выбрать* для занесения и выборки пакетов данных. Хотя его спецификации интерфейса аналогичны спецификациям других используемых в системе буферных задач, его реализация, как мы увидим позднее, имеет существенные отличия:

```
--Буфер пакета данных
with тип_пакет_данных;
use тип_пакет_данных;
package пакетный_буфер is
  task буферизация is
    entry занести(dp : in пакет_данных);
    entry выбрать(dp : out пакет_данных);
  end буферизация;
end пакетный;
```

13.3.3. Реализация

Приведенные выше спецификации обеспечивают модульную структуру системы, для которой были четко определены все характеристики интерфейса. Все, что теперь остается, — это разработать тело каждой из компонент, так чтобы предполагаемые на стадии спецификации функции были должным образом реализованы. Прежде чем привести вариант каждой из компонент, следует подчеркнуть, что в реальном проектировании программного обеспечения с помощью языка Ада приведенные выше спецификации были бы, вероятно, разработаны старшим программистом, который затем бы раздал спецификации отдельных компонент различным членам его группы разработки. Так как интерфейс между отдельными компонентами уже определен и так как все компоненты можно компилировать самостоятельно, каждый член группы может разрабатывать, проверять и отлаживать свою часть системы независимо от других. Эта способность поддерживать групповую разработку была одной из основных задач языка Ада.

Вернемся к примеру нашей программы; каждая компонента будет описываться в том порядке, в котором приводились ее спецификации.

Тело пакета *тип_пакет_данных* имеет следующий вид:

```
package body тип_пакет_данных is
  function преобразовать(dp : пакет _данных) return строка is
  --формат пакета данных в коде ASCII имеет вид
  --ccdddhhtnmssnnvvvv...nnvvvv
   --где
  -cc                  = число выборок в пакете
  -dddhhmmss       = время, когда были взяты выборки
   --nnvvvv           =значение выборки vvvv взято из канала пп
     use тип_время;
     длина : constant естеств := 11 + 6*dp.размер;
     послед_pkt : строка(1..длина);
     индекс : целый range 1..длина := 12;
     function числоcvt(номер : позцел; размер : естеств) return
                                                                    строка;
        s : строка(1..размер);
        знач : позцел := номер;
        нуль : позцел := символьный'поз('0');
      begin
       for i in reverse 1..размер loop
          s(i) := символьный'знач(знач mod 10 + нуль);
          знач := знач/10;
        end loop;
        return s;
      end число cvt;
    begin
      послед_pkt(1..2) := числоcvt(dp.размер, 2);
      послед_pkt(3..11) := числоcvt (dp.t, 9);
      for i in 1..dp.размер loop
        послед_pkt(индекс. .индекс + 1) :=
                                              числоcvt(dp.данные(i).cn,
       2);
        послед_pkt(индекс + 2..индекс + 5) :=
                                         числоcvt(dp.данные(i).cr, 4);
        индекс := индекс + 6;
       end loop;
       return послед-pkt;
     end преобразовать;
  end тип_пакет_данных;
```

В теле содержится только функция *преобразовать* для отображения внутреннего представления пакета данных на внешнее представление в виде строки символов. Обратите внимание на использование динамического регулярного типа *строка* и на присваивания вырезок из массива при построении последовательного потока символов.

Тело пакета *тип_пакет_управления* тоже содержит одну функцию преобразования. Код этого тела имеет следующий вид:

```
package body тип_пакет_управления is
  function преобразовать(s. строка) return пакет_управления is
  --формат пакета управления в коде ASCII
  --
  --onnsss        Открыть канал пп с периодом sss
  --annsss        Изменить период канала пп на sss
  --cnn           Закрыть канал пп
  --tdddhhmmss    Установить время
   use тип_время;
   длина : естественный := s'длина;
   кмд : команда;
   type длины_пакет is array (команда) of естественный;
   кор_pkt_длины: constant длины_пакет :=
                              (открыть_канал = > 6,
                               изменить_период = > 6,
                               закрьть_канал = > 3,
                               установить, время = >10);
   function кмдcvt(c : символьный) return команда is
   begin
      case c is
         when 'o' = > return открыть_канал;
         when 'a' => return изменить_период;
         when 'c' => return закрыть_канал;
         when  't'=> return установить_время;
         when others => raise ср_формат_ош;
      end case;
    end кмдcvt;
   function номерcvt(номер : строка) return позцел is
      знач : позцел := 0;
      нуль : constant позцел := символьный'поз('0');
   begin
      for i in reverse номер'диапазон loop
         if номер(i) in '0'..'9' then
            знач := знач* 10 + символьный'поз(номер(i)) – нуль;
         else
            raise ср_формат_ош;
         end if;
      end loop;
      return знач;
    end номерcvt;
begin
   кмд := кмдcvt(s(l));
   if длина < > знач_pkt_длина(кмд) then
      raise ср_формат_ош;
   end if;
   case кмд is
      when открыть_канал | изменить_период =>
           return (кмд, номерcvt(s(2..3)), номерcvt(s(4..6)));
      when закрыть_канал =>
          return (кмд, номерcvt(s(2..3)), 0)
       when установить_время =>
          returned, отоб_время(8(2..10)));
    end case;
  exception
    when others => raise ср_формат_ош;
  end преобразовать;
end тип_пакет_управления;
```

Кроме того очевидного факта, что отображение осуществляется в противоположном направлении, т. е. из внешнего формата во внутренний, следует отметить и наличие добавочной проверки, которая необходима в случае неправильного форматного задания вводного управляющего пакета. Исключение *ср_формат_ош* используется для фиксирования такой ситуации. Заметьте, как ненавязчиво работает механизм исключения. Команда проверки ошибки никак не нарушает потока управления главной программы.

Обе основные подсистемы специфицируются несколько иначе, чем составляющие задачи специфицируются внутри пакета. Язык Ада требует, чтобы тело пакета, подпрограммы или задачи находилось в той же самой описательной части. Впрочем, на месте тела может быть помещена заглушка (stub), так что фактическое тело может быть описано отдельно. Каждая подкомпонента должна описать в своем заголовке имя своей родительской компоненты. Поэтому все следующие шесть подкомпонент содержат предложение

```
separate (подсистема_связи)
```

в качестве префикса.

Стандартным примером на языке Ада является настраиваемое тело пакета *сим_буфер* — классический ограниченный буфер. Приводим его без комментариев.

```
separate (подсистема_связи)
package body сим_буфер is
    task body буферизация is
      буфер : array(1..размер) of символьный;
      счетчик : позцел := 0;
      хвв, хиз : естественный := 1;
    begin
        loop
           select
              when счетчик (размер = >
                   accept занести(с: in символьный) do
                      буфер(хвв) := с;
                   end;
                   хвв := хвв mod размер + 1; счетчик := счетчик + 1;
           or
              when счетчик >0 =>
                   accept выбрать(с : out символьный) do
                       с := буфер(хиз);
                   end;
                   хиз := хиз mod размер + 1; счетчик := счетчик - 1;
           end select;
        end loop;
     end буферизация;
  end сим_буфер;
```

Обе задачи управления устройствами *tx_драйвер* и *rx_драйвер* также достаточно просты. Впрочем, отметим, что регистр статуса специфицирован типом *целый.* Концептуально это не очень привлекательно. Было бы лучше использовать некоторую форму битовой строки, но спецификацию представления Ада предусматривает только для записей, но не для массивов (см. задачу АЦП). Поэтому «битовый» тип может быть введен только с помощью указания, например

```
type биты is array(0..15) of логический;
pragma упаковать(биты);
```

Но спецификации языка не гарантируют, что указание обеспечит требуемый нам результат, т. е. упакует весь массив в одно 16-битовое слово, где индекс 0 будет обозначать самый младший бит. Следовательно, в этом отношении язык Ада следует признать несбалансированным языком.

```
separate (подсистема_связи)
task body tx_драиеер is
   сиз : символьный;
   статус : целый;
   буфер : символьный;
   for статус use at tx_сма;
   for буфер use at tx_буф;
begin
   loop
      буфер := сиз;
      статус := ввести_нач;
      accept прерывание;
      статус := снять_нач;
   end loop;
 end tx_драйвер;
 separate (подсистема_связи)
 task body rx_драйвер is
   сев : символьный;
   статус: целый;
  буфер : символьный;
 for статус use at rx_сма;
   for буфер use at rx_буф;
  begin
     loop
        статус := ввести_нач;
        accept прерывание;
        статус := снять_нач;
        сев := буфер;
        rx_занести(свв);
     end loop;
   end rx_драйвер;
```

Задача *rx_управление* реализует взаимно исключающий доступ к буферу вывода. Она кодируется следующим образом:

```
with системно_глобальный; use системно_глобальный;
separate (подсистема_связи)
task body tx_управление is
begin
   loop
     accept оградить;
     передача;
     loop
        select
          accept tx(c : in символьный) do
             tx_занести(c);
          end;
        or
          accept освободить(re1_класс : in класс_освободить;
                  ок : out логический) do
             if re1_класс = после_под then
                select
                  accept ответ_получен(подтверждсн : in
                                                    логический) do
                    ок := подтвержден:
                  end;
                or
                  delay глав_связь_задержка;
                  ок := ложь;
                end select;
             else
                ок := истина;
             end if;
          end освободить;
          exit передача;
        end select;
     end loop передача;
   end loop;
 end tx_управление;
```

Интересной особенностью этой задачи является то, как она использует вложенные операторы приема. Когда задача обращается к входу *освободить,* указывая, что она будет ждать подтверждения от главной машины, то она задерживается на входе *освободить,* в то время как задача *tx_управление* выполняет внутренний оператор приема. Поэтому обращающаяся к входу *освободить* задача получит разрешение продолжать работу либо когда произойдет обращение к входу *ответ_получен,* либо когда истечет время задержки.

Задача *tx_интерфейс* тоже пишется очень просто. Она циклически выбирает пакет данных из буфера *пакетный_буфер,* преобразовывает его в последовательную форму и затем передает его. Если параметр *ок* возвращается задаче *tx_управленинue_освободить* со значением *ложь,* то пакет передается повторно.

```
with тип_пакет_данных, пакетный_буфер;
use тип_пакет_данных, пакетный_буфер;
separate (подсистема_связи)
task body tx_интерфейс is
  dp : пакет_данных;
begin
   loop
      буферизация.выбрать(dp);   --из пакетного буфера
      declare
         последовательный_pkt : constant строка := преобразо-
                                                       вать(dp);
        ок: логический;
        консум : целый range 0.. 127;
        сиз: символьный;
      begin
         loop
            консум := 0;
            tx_управление.оградить;   --потребовать связь tx
            tx_управление.tx(stx);
            for i in последовательный_pkt'диапазон loop      --послать
                                                           пакет
               сиз := последовательный_pkt(i);
               консум := (консум + символьный'поз(сиз)) mod  128;
               tx_ управление.tx(сиз);
            end loop;
            tx_управление.tx(символьный'знач(консум));
            tx_управление.tx(etx);
            tx_управление.освободить(после_под, ок);   -освободить
                                                     связь tx
            exit when ок;
          end loop;
       end;
    end loop;
end tx_интерфейс;
```

Обратите внимание на использование внутреннего описания в блоке. Это необходимо, так как неизвестен размер пакета *последовательный_pkt* до тех пор, пока внутренний *пакет_данных* не будет извлечен из *пакетный_буфер.*

Задача *rx_интерфейс* кодируется гораздо более сложным образом, так как она должна обрабатывать наряду с поступающими пакетами управления символы подтверждения ACK/NAK. Код этой задачи имеет следующий вид:

```
with тип_пакет_управления, подсистема_сканирования;
use тип_пакет_управления, подсистема_сканирования;
separate (подсистема_связи)
task body rx_интерфейс is
   свв : символьный;
   длина : естественный;
   максдлина : constant :== 20;
   ввод : строка(1..максдлина);
   ввод_ош: exception;

procedure читать_пакет(l : out естественный) is
   консум: целый range 0. .127 := 0;
   сев: символьный;
   индекс :  целый range 0.. максдлина := 0;
begin
   loop
     if индекс >= максдлина then raise ввод_ош;
     rx_ выбрать(свв);
     exit when свв = etx;
     индекс := индекс + 1;
     ввод(индекс) := индекс;
   end loop;
   for i in 1..индекс - 1 loop
     консум := (консум + символьный'поз(ввод(i))) mod 128;
   end loop;
   if консум < > символьный'поз(ввод(индекс)) then
     raise ввод.ош;
   end if;
   l := индекс - 1;
end читать_пакет;
```

```
procedure послать_ответ(c : in символьный) is
begin
  tx_управление.оградить;
  tx_управление.tx(c);
  tx_управление.освободить(немедленно);
end послать_ответ;

procedure задержать_глав_ответ(подтвержден: in логический) is
begin
    select --ответ задержать, только если tx_управление его ждет
       tx_управление.ответ_получен(подтвержден);
   else
       null;
    end select;
end задержать_глав_ответ;

begin
    loop
       loop   --пропустить все лишнее в вводном потоке
          rx_выбрать(свв);
           exit when свв = stx or свв = под or свв = nak;
       end loop;
       if свв = stx then
          begin
             читать_пакет(длина);
             declare
                s : constant строка := ввод(1..длина);
                срх: пакет_управления;
             begin
                срх : преобразовать(s);   --преобразовать строку
                                            в пакет управления
                обновить(срх);              --послать пакет программе
                                      -- управления таблицей
                                      --сканирования
             end;
             послать_ответ(под);
           exception
             when others =>
                послать_ответ(nak);
           end;
        els if сев = под then
          задержать_глав_ответ(истина);
        else
          задержать_глав_ответ(ложь);
        end if;
   end loop;
end rx_интерфейс;
```

В этой задаче интересно отметить включение условного обращения к выходу в процедуре *Задержать_глав_ответ.* Это позволяет информировать задачу *tx_управление* о получении подтверждения главной машины только в том случае, когда она на самом деле ждет этого ответа. Это предотвращает тупиковую ситуацию в системе при получении сигналов ACK/NAK после срабатывания задачи *глав_связь_задержка.*

Этим завершается разработка подсистемы связи.

Система сканирования состоит из трех задач. Программа управления таблицей сканирования следит за состоянием таблицы сканирования, которая содержит статус и периоды выборки для каждого измерительного канала. Сама таблица сканирования реализуется как вложенный в тело задачи пакет. Этот пакет предоставляет четыре процедуры для доступа к таблице, но саму таблицу от задачи прячет. Поэтому задача исполняет простой цикл, принимая либо обращение для обновления таблицы, либо обращение для поиска в таблице. Ее код имеет следующий вид:

```
separate (подсистема_сканирования)
task body управление_таблицей_сканирования is

  package таблица_скан is
    procedure закрыть(n : in номер_канала);
    procedure открыть(n : in номер_канала;  p : in период);
    procedure изменить(n : in номер_канала; новр : in период);
    procedure проверить (общий : out позцел; готово : out
               готово_уст; tтеперь : out время);
  end таблица_скан;

  package body таблща_скан is
    use системные_часы;
     type канал_запись is
       record
         активный : логический := ложь;
         когда: время;
         период_выборки : период;
       end record;
    таблица : array (номер_канала) of канал_запись;

    procedure закрыть(n : in номер_канала) is
    begin
       таблица(n).активный := ложь;
    end;

    procedure открыть(n : in номер_канала; p: in период) is
    begin
       таблица(n) := (истина, время_теперь( ), p);
    end;

    procedure изменить(n : in номер_канала; новр : in период) is
    begin
      таблица(n).период_выборки := новр;
    end;
```

```
      procedure проверить(общий : out : позцел : готово : out
               готово_уст; tтеперь : out время) is
         счетчик : позцел := 0;
         t : время := время_теперь( );
      begin
         for n in номер_канала loop
            declare
               с : канал_запись renames таблица(n);
            begin
               if с.активный and t > с.когда then
                  с.когда := с.когда + с.период_выборки;
                  готово(n) := истина; счетчик := счетчик + 1;
               else
                  готово(n) :== ложь;
               end if;
            end;
         end loop;
         tтеперь := t; общий := счетчик;
      end проверить;

end таблицамкан;

срх : пакет_управления;
use таблица_скан;
begin
   loop
      select
         accept обновить(ср : in пакет_управления) do
            срх := ср;
         end;
         case срх.кмд is
            when открыть_канал => открыть(срх.сп, срх.р);
            when закрыть_канал => закрыть(срх.сп);
            when изменитъ_период => изменить(срх.сп, срх.р);
            when уст_время => системные_часы.время (срх.t);
         end case;
      or
         accept проверить(общий : out позцел; готово : out
                              готово_уст; t: out время) do
            проверить(общий, готово, t);
         end;
      end select;
   end loop;
end управление_таблицей_сканирования;
```

Задача драйвера АЦП иллюстрирует использование спецификаций представления в языке Ада.

```
separate(подсистема_сканирования)
task body ацп is
  type бит is (нет, есть);
  type статус_регистр is
    record
       старт: бит;
       inten : бит;
        кан : номер_канал;
     end record;
   статус_рег: статус_регистр;
   буфер_рег: считывание_канал;
   for статус_регистр use
      record
         старт at 0 range 0..0;
         inten at 0 range 6.. 6;
         кан at 0 range 10.. 15;
      end record;
   for статус_рег use at ацк_ста;
   for буфер_рег use at ацк_буф;
begin
  loop
    accept читать(n: in номер_канала; r : out считывание_канал) do
       статус_рег := (есть, есть, n);
       accept преобразование_завершен;
       статус_рег := (нет, нет, 0);
       r := буфер_рег;
    end читать;
  end loop;
end ацп;
```

Регистр статуса АЦП содержит два бита для запуска преобразования и обеспечения прерываний и поле из шести битов для спецификации номера канала. Поэтому используемой структурой является запись, и отображение каждой компоненты записи явно специфицируется в спецификации представления. Сама задача состоит из цикла, который принимает вход *читать*. Впрочем, и здесь обратите внимание на то, как используется вложенный прием для сдерживания первой задачи до тех пор, пока не произойдет некоторое другое событие. В данном случае таким событием является прерывание.

Последней задачей в подсистеме сканирования является программа расписания. Функционирование этой задачи достаточно просто *к* очевидно, но опять обратите внимание на исполь-

зование вложенного блока описания, вводящего динамический объект пакета данных *dpx*.

```
with тип_пакет_данных, пакетный_буфер;
use тип_пакет_данных, пакетный_буфер;
separate (подсистема_ сканирования)
task body расписание is
  общий_готово : целый range 0..чис_каналов;
  готово : готово_уст;
  t : время;
begin
   loop
     delay распис_цикл_время;
     управление_таблицей_сканирования.проверить(общий_готово,
     готово, t);
     if общий_готово > 0 then
        declare
          dpx : пакет_данных(общий_готово);
          индекс : целый range 1..общий_готово := 1;
         begin
           dpx.t := t;
           for i in номер_канала loop
             if готово(i) then
                ацп.читать(i, dpx.данные(индекс).cr);
                dpx. данные(индекс)..cn := i;
                индекс := индекс + 1;
             end if;
           end цикл;
           буферизация.занести(dpx);
         end;
        end if;
     end loop;
  end расписание;
```

Единственной оставшейся компонентой, которую нам нужно реализовать, является пакетный буфер. Его нужно построить так, чтобы имелась возможность использовать всю оставшуюся память для буферизации как можно большего числа пакетов. Поэтому используется динамический тип данных. Сам буфер является связанным линейным списком, пакеты присоединяются к концу списка до тех пор, пока не исчерпывается вся память. Только после этого происходит блокирование поступающих пакетов. Код для пакетного буфера имеет следующий вид:

```
with непроверяемое_распределение;
package body пакетный_буфер is
  task body буферизация is

  package буфер is
     function полон return логический;
     function пуст return логический;
     procedure присоединить(dp : in пакет_данных);
     procedure удалить(dp : out пакет_данных);
  end буфер;

package body буфер is
   type ячейка(dpразмер : целый range 1..чис_каналов);
   type звено is access ячейка;
   type ячейка(dpразмер : целый range 1..чис_каналов);
      record
         dp: пакет_данных(dpразмер);
          следующий : звено;
      end record;
   голова, хвост : звено;
   ждущий_пакет : пакет_данных;
   память_полна : логический;
   procedure свободный is new непроверяемое_распределе~
                 ние(ячейка, звено);
   function полон return логический is
   begin
      return память_полна;
   end;
   function пуст return логический is
   begin
      return голова = null;
   end;
   procedure присоединить(dp: in пакет_данных) is
      врем : звено;
   begin
      врем := new ячейка(dp.размер);  --создать ячейку
                                       --с ограничением
      врем.dp :=  dp;                  --копировать в нее
                                       --пакет
      врем.следующий := null;
      if хвост = null then             -- соединить со
                                       --списком
        голова :=  врем; хвост :=  врем;
      else
        хвост.следующий := врем; хвост :=  врем;
      end if;
    exception
      when ошибка_памяти =)
         ждущий_пакет := dp;
         память_полна := истина;
    end присоединить;
```

```
        procedure удалить(dp : out пакет_данных) is
            врем : звено := голова;
    begin
            dp : врем.dp;
            if голова = хвост then
                голова := null; хвост := null;
            else
                голова := голова. следующий;
            end if;
            свободный(врем);     --разместить удаленную ячейки
            if память_полна then
                память_полна := ложь;  --попытаться поместить на
                                       --буфер ждущий пакет
                присоединить(ждущий_пакет);
            end if;
        end удалить;
      use буфер;
        begin
          loop
            select
              when not полон = >
                accept занести(dp : in пакет_данных) do
                  присоединить (dp) ;
                end;
            or
              when not пуст = >
                accept выбрать(dp : out пакет_данных) do
                  удалить(dp);
                end;
            end select;
          end loop;
        end буферизация;
  end пакет_буферизация;
```

Как легко увидеть, фактически буфер реализуется в виде пакета, вложенного в задачу буферизации. Интерфейс между буфером и задачей осуществляется с помощью двух процедур, которые присоединяют и удаляют пакеты к буферу и из буфера, и двух функций, проверяющих, является ли буфер пустым или заполненным. Сама задача имеет точно такую же форму, как и приведенная ранее задача символьного буфера.

При построении пакетного буфера используется присоединение динамически распределяемых ячеек к списковой структуре. Когда происходит обращение к процедуре *присоединить,* создается новая ячейка и в нее заносится поступивший пакет. Затем новая ячейка присоединяется к концу списка, на который

указывает статическая переменная доступа *хвост.* Когда главная вычислительная машина работает временно в автономном режиме, пакетный буфер будет расти до тех пор, пока не исчерпается вся имеющаяся память. Когда это произойдет, обращение к процедуре *присоединить* вызовет возбуждение исключения *ошибка_памяти.* В этом случае поступающий пакет запомнится в переменной *ждущий_пакет* и произойдет установка флажка *память_полна,* что заблокирует все дальнейшие попытки разместить на буфере новый пакет. Процедура *удалить* выбирает первую ячейку из списка, на которую указывает переменная доступа *голова* и возвращает удаленный пакет вызывающей программе. Затем процедура освобождает память, занимаемую удаленной ячейкой. Если флажок *память_полна* установлен, процедура *удалить* пытается также присоединить ждущий пакет к буферу, так чтобы можно было возобновить обычные буферные операции.

Следует заметить, что это решение предполагает наличие механизма полного динамического распределения и перераспределения памяти. Как было замечено в гл. 3, может оказаться, что такая система будет функционировать недостаточно быстро, и хотя это не является большой проблемой в нашем примере, для многих приложений реального времени такое решение может оказаться неприемлемым. В связи с этим язык Ада разрешает программисту специфицировать максимальную память, которая должна быть отведена для набора динамических объектов, в относящейся к длине спецификации представления; это дает возможность использовать более простые и более эффективные стратегии управления стековой памятью. При фактической реализации языка Ада такая спецификация длины может требоваться обязательно, и в этих случаях формулировку пакетного буфера придется модифицировать.

13.4. ОБСУЖДЕНИЕ

Из предыдущего описания языка и разобранной программы должно стать ясно, что язык Ада — очень большой и содержательный язык. Действительно, если отвлечься от отсутствия типа множественных данных, Ада обеспечивает все основные возможности, которые были идентифицированы в гл. 9 как необходимые для современного языка реального времени общего назначения. При обсуждении языка Ада нужно рассмотреть два различных аспекта. Во-первых, имеются общие соображения, относящиеся в целом к предоставлению средства программирования в реальном времени, и, во-вторых, более конкретные соображения, касающиеся деталей индивидуальных особенностей самого языка. Вне контекста данной книги такое

обсуждение неизбежно заняло бы очень много места. Но очевидно, что большинство рассуждений из первой части книги непосредственно применимы и к языку Ада. Поэтому в дальнейшем мы ограничимся обсуждением тех моментов, которые не были рассмотрены ранее.

На самом общем уровне наиболее заметным и с первого взгляда наиболее пугающим аспектом языка Ада являются его общий объем и сложность. С точки зрения потенциальных пользователей языка Ада, из этого следует сделать три главных вывода: будет трудно обучать программистов, разработка программного обеспечения будет зависеть от методов кросс-компиляции [11] система прогона написанных на языке Ада задач будет очень большой.

Обучение программиста до достаточного уровня компетентности в рамках полного объема языка Ада может оказаться еще более трудной задачей, чем та, которую ставит сам объем языка. Действительная трудность связана не только с количеством предоставляемых возможностей, сколько с природой этих возможностей. Механизмы пакетов и задач языка Ада в сочетании с исчерпывающей системой очень сильной типизации требуют использования методологии программирования, которая может оказаться совершенно чуждой для среднего программиста, привыкшего к традиционным языкам высокого уровня. Поэтому проблема сводится не просто к изучению нового языка, но к тому, как научиться писать программы новым способом. Но само по себе это не так уж и плохо. Одной из главных мотивировок создания языка Ада было прежде всего значительное усовершенствование существующей практики разработки и реализации программного обеспечения. Ясно, что шаг вперед в этом направлении неизбежно связан с большими расходами по переучиванию.

Необходимость использования методов кросс-компиляции является прямым следствием неизбежной сложности компиляторов для языка Ада. В настоящее время точно еще неизвестно, насколько большие вычислительные машины потребуются для поддержки программирования на языке Ада, но, по-видимому, приемлемую скорость компиляции можно будет получить лишь на больших вычислительных машинах. Из этого следует, что пользователям с небольшими запросами возможно лотребуется работать при программировании на языке Ада на удаленных консольных установках. В тех случаях, когда это окажется

[11] Компиляция на одной (обычно большей) ЭВМ машинного кода для другой вычислительной системы (обычно для микро-ЭВМ).—*Прим. ред*

именно так, недостатки удаленной разработки не смогут компенсироваться достоинствами языка, большинство которых окажутся потерянными. Впрочем, можно надеяться, что быстрое развитие микропроцессорной технологии с одновременным совершенствованием искусства написания трансляторов обеспечит достаточно недорогое производство небольших рабочих станций с встроенным языком Ада. Поэтому и для пользователей с ограниченными запросами использование разработки на удаленной главной вычислительной машине является проблемой, которая должна отпасть в недалеком будущем.

Сложность механизма задач языка Ада и поддержки, необходимой для обработки исключений, проверки диапазонов, проверки ограничений и управления памятью, означает, что получаемый объектный код будет относительно громоздким. Безусловно, необходимое для поддержки механизма задач ядро будет во много раз больше, чем описанное в предыдущем разделе ядро для языка Модула. Кроме этого, в Аде потребуется гораздо больше проверочного кода в поточном режиме, чем, скажем, в Паскале, так как границы поддиапазона и границы индексов массивов могут быть динамическими. Следовательно, маловероятно, что язык Ада будет экономичен для небольших, базирующихся на микропроцессорах систем, по крайней мере в ближайшем будущем.

Рассмотрим теперь конкретные особенности, которые предоставляет язык Ада; наиболее важной одиночной конструкцией языка Ада является, надо полагать, пакет. Определение пакета здесь отличается довольно существенно от определения, рассмотренного в гл. 5. Прежде всего в Аде довольно свободные правила открытой области действия. Поэтому объекты, описанные вне пакета, но видимые в точке его описания, автоматически видимы и внутри пакета. Следовательно, пакет является не глухой стеной области действия, как в Модуле, а односторонней мембраной. Обладая тем преимуществом, что сокращается текстуальная длина спецификаций пакета, это свойство ограничивает эффективность спецификаций пакета как средства интерфейса. Примером такого ограничения является пакет *подсистема_связи* из предыдущего раздела. В этом пакете спецификация пуста, пакет общается с остальной частью системы только с помощью односторонних процедур и обращений ко входам. Поэтому, чтобы понять, как его функционирование зависит от других пакетов, необходимо детальное рассмотрение его реализационной части. По-видимому, это следует считать серьезным недостатком пакета языка Ада.

Если пакет содержит определение приватных типов, то все подробности типа должны быть приведены в приватной части спецификации пакета Необходимость этого связана с возмож-

ностью раздельной компиляции, что подробно разбиралось в гл. 5. Хотя простых альтернатив этого решения не видно, эта особенность тем не менее достаточно неприятна. Во-первых, она предоставляет пользователю пакета информацию, знать которую у него нет никакого права, и, во-вторых, это означает, что изменения в реализации приватного типа могут потребовать изменений в приватной части спецификации. Это противоречит принципу наличия отдельных спецификационных и реализационных частей. Поэтому жаль, что разработчики Ады не смогли найти более удовлетворительной альтернативы.

Механизм задач в языке Ада и элегантен, и достаточно мощен; хотя, как и в случае пакетов, из принятых правил области действия вытекает ряд неприятных следствий. У активных задач (т. е. у тех, у которых нет входов) спецификационная часть пуста, поэтому и здесь их интерфейс со всей остальной частью системы можно определить, только исследовав их реализационные части. Кроме того, и это значительно более серьезно, нет никаких ограничений на использование общих переменных. Обеспечение взаимного исключения при использовании входов связано в этих условиях с достаточно тонкой проблемой надежности. При обращении к входу параметры вычисляются перед вхождением в рандеву. Это означает, что две задачи могут обратиться к входу одновременно, объявляя в качестве параметра **in out** одну общую переменную. Если этот параметр используется как ключ доступа к некоторому ресурсу, то обе задачи могут получить доступ к этому ресурсу одновременно, так как начальное значение ключа копируется на вход перед обращением к рандеву (см. Уэлш и Листер, 1981, где этот эффект объясняется полностью). И последнее замечание, касающееся задач: как и в языке Модула, нельзя написать программу расписания, которая бы запускала и останавливала другие задачи без их взаимодействия. Управление задачами ограничивается операциями завершения *отказ/прекращение.*

Язык Ада предоставляет богатый набор возможностей для структурирования данных, хотя отсутствие множественного типа в некоторых приложениях может вызвать сожаление. Возможность задавать компонентам записи начальное значение в описании типа является крайне полезным свойством при реализации абстрактных типов данных. Впрочем, не совсем ясно, почему эту возможность нужно было ограничивать только записями. Безусловно, необходимость вводить комбинированный тип с одной компонентой только для того, чтобы задать скалярному типу значение по умолчанию, несколько раздражает.

Динамические массивы и дискриминантные записи могут быть очень полезными, это можно увидеть в примере нашей программы при обработке пакетов данных переменной длины.

Однако на их использование язык Ада накладывает несколько очень сильных ограничений. В случае статического комбинированного объекта дискриминант можно изменить, только присваивая новое значение всей записи. Там, где дискриминантными являются вариантные записи, это является существенным средством обеспечения надежности (см. разд. 3.3.2), но там, где дискриминантны границы массива, это ограничение не кажется столь необходимым. И на самом деле, относящиеся к дискриминантам правила были бы проще и могли бы быть менее жесткими, если бы производилось различие между границами индексов и дискриминантами вариантного случая. В случае динамически распределяемых объектов дискриминант всегда должен быть ограничен и всегда должен оставаться неизменным. По-видимому, это правило тоже без необходимости слишком строго.

Возможно, что самой слабой чертой языка Ада являются его средства программирования низкого уровня. Эти средства удивительно несбалансированы и в некоторых отношениях недостаточны. В предыдущем разделе была уже упомянута проблема определения в Аде *битового* типа. Для перечислимых и комбинированных типов Ада предоставляет только спецификации представления, поэтому отображение массива логических значений в языке точно специфицировано быть не может. Обработка прерываний осуществляется с помощью связи входа задачи с векторным адресом. В принципиальном отношении это аналогично Модуле; но в отличие от Модулы в Аде нет никакого механизма для работы с приоритетами прерываний. Предоставляется указание для установки приоритетов задач, но этот механизм разрешен для любых задач и, по-видимому, предназначался в первую очередь для установки приоритетов программного обеспечения, а не машинного оборудования. Действительно, в справочнике языка прямо говорится, что прерывание трактуется так, как если бы оно было обращением к входу от задачи, приоритет которой превосходит приоритет любой другой определенной пользователем задачи. Из этого следует, что прерывания с низким приоритетом могут прерывать выполнение задачи, обслуживающей прерывание более высокого уровня! Доступ к регистрам устройств в Аде можно осуществить, либо используя процедуры ввода-вывода низкого уровня *послать_управление* и *получить_управление,* либо отображая переменные на регистры адресов памяти, как в примере нашей программы. Однако вторая возможность должна использоваться так, чтобы не возникала необходимость точно определять последствия присваивания отображенной переменной (Хиббард и др., 1980). Это, по-видимому, сильно обесценивает наличие возможности спецификаций представления для комбинированных

типов. В заключение можно сказать, что общее впечатление от предоставляемых языком Ада возможностей в этой области таково, как будто разработчики языка на самом деле и не хотели поддерживать программирование низкого уровня непосредственно из языка.

Следует подчеркнуть, что данные комментарии и критические замечания относительно языка Ада основываются на первом впечатлении. Поэтому может оказаться, что некоторые из них необоснованы, и, безусловно, выявится много других недостатков, когда накопится достаточно опыта практического использования языка. Но, разумеется, их не следует считать следствием недостаточного энтузиазма по отношению к самому языку. В целом язык Ада является превосходным примером проекта языка программирования. В четкой и единой языковой -схеме он объединяет самые лучшие концепции последнего десятилетия. Его размер и сложность являются неизбежным следствием заданного набора требований, которые пытались удовлетворить его разработчики. В конце концов язык Ада — это язык профессионалов, предназначенный для написания высококачественного программного обеспечения. Труд и настойчивость, которые требуются от программиста, изучающего и использующего этот язык, пойдут ему только на пользу; он не только изучит язык Ада — он также научиться конструировать свои программы должным образом,

ЛИТЕРАТУРА

Ахо, Ульман (Aho A. V., Ullman J. D.)
(1978) Principles of Compiler Design, Addison Wesley. [Имеется перевод: Ахо А. В., Ульман Д. Д. Теория синтаксического анализа, перевода и компиляции. — М.: Мир, 1978.]

Барнес (Barnes J. G. P.)
(1972) Real Time Languages for Process Control, Computer J., **15**, p. 15-17.
(1976)RTL/2; Design and Philosophy, Heyden, London.

Брандес, Эйхентопф (Brandes J., Eichentopf S. et al.)
(1970) PEARL: The Concept of a Process-and Experiment-oriented Programming Language, Elektronische Dataverarbeitung, **10, 12**, p. 162-175.

Брон, Фоккинга (Bron C., Fokkinga M. M.)
(1977) Exchanging Robustness of a Program for a Relaxation in its Specification, Memo 178, Dept of Applied Maths, Twente University of Technology, Netherlands.

Ванд (Wand I. C.)
(1980) Dynamic Resource Allocation and Supervision with the, programming; language Modula, Computer J., 23, 2, p. 147—152.

Вейнгаарден (Van Wijngaarden A., et al)
(1975) Revised Report on the algorithmic language Algol 68, Acta Informatica, 5 [Имеется перевод: Вейнгаарден А. Пересмотренное сообщение об АЛГОЛЕ 68.—М.: Мир, 1979.]

Вирт (Wirth N.)
(1971a) The Programming Language Pascal, Acta Informatica, **1**, p. 35—63.
(1971b) Program Development by Stepwise Refinement, Comm ACM, **14**, 4.
(1977a) Modula: a language for modula multiprogramming, Software P and E, 7, p. 3.
(1977b) The Use of Modula, Software P and E, 7, p. 37.
(1977c) Design and Implementation of Modula, Software P and E, **7**, p. 67.
(1980) Modula 2, Report 36, Institut fur Informatik, ETH, Zurich.

Вихман (Wichmann B. A.)
(1979) Integer Division, Software P and E, **9**, p. 507—508.

Вудвард, Везеролл, Горман (Woodward P. M., Wetherall P. R., Gorman B)
(1970) Official Definition of Coral 66, HMSO,. London.

Гешке, Морис, Сатертвайте (Geschke C M., Morris J. H., Jr, Satterthwaite E. H.)
(1977) Early Experience with Mesa, Comm ACM, 20, 8, p. 540—552.

Гешке, Сатертвайте (Geschke C. M., Satterthwaite E. H.)
(1977) Exception Handling in Mesa, Xerox Pare Report, Palo Alto.
Грис (Gries D.)
(1971) Compiler Construction for Digital Computers, Wiley, London and New York. [Имеется перевод: Грис. Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975.]
Гудинаф (Goodenough J. B.)
(1975) Exception Handling: Issues and a proposed notation, Comm ACM, 18, 12, p. 683—696.
Даглес, Даусинг (Dagless E. L., Dowsing R.)
(1979) Design Methods for Digital Systems —Part 1: Concurrency Constructs, IEE, CDT, 2, 3, p. 93—99.
Дал, Мюрхауг, Нюгорд (Dahl O-J., Myhrhaug B., Nygaard K.)
(1968) The Simula 67 Common Base Language, Norwegian Computer Centre, Oslo. [Имеется перевод: Дал У.-И., Мюрхауг Б., Нюгорд К. Симула 67. Универсальный язык программирования. — М.: Мир, 1969.]
Даусон (Dowson M., et al.)
(1979) The DEMOS Multiple Processor, Euro IFIP 79, Ed. P. A. Samet, North Holland.
Дейкстра (Dijkstra E. W.)
(1968) Co-operating Sequential Processes, in Programming Languages, Ed. F. Genuys, Academic Press. [Имеется перевод: Дейкстра Э. Взаимодействие последовательных процессов. В кн.: «Языки программирования> под редакцией Ф. Женюи. — М.: Мир, 1972.]
(1972) Notes on Structured Programming, in Structured Programming, Academic Press. [Имеется перевод: Э. Дейкстра. Заметки по структурному программированию. В кн.: У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975.]
Джемисон (Jamieson M. J.)
(1980) Integer Division, (letter), Software P and E, **10**, p. 333.
Ихбиа (Ichbiah J. D.)
(1978) Preliminary Reference Manual for the Green Language, CII-Honeywell Bull, Louveciennes, France.
(1979) Reference Manual for the Green Programming Language, CII-Honeywel! Bull., Louvecinnes, France. (Also publ. as Preliminary Ada Reference Manual, Sigplan Notices **14**, 6, 1979.)
(1980) Reference Manual for the Ada Programming Language, United States Dept. of Defense.
Ихбиа, Ферран (Ichbiah J. D., Ferran G.)
(1977) Separate Definition and Compilation in LIS and its Implementation, Cornell Symposium on High Order Languages, in Lecture Notes on Computer Science, Springer-Verlag.
Ихбиа, Риссен, Хелиард, Кусо (Ichbiah J. D., Rissen J. P., Heliard J. C., Cousot P.)
(1974) The System Implementation Language LIS, Reference Manual CII-HB Technical Report 4549E/EN, CII-Honeywell Bull, Louveciennes, France.
Йенсен. Вирт (Jensen K., Wirth N.)
(1979) Pascal User Manual and Report, 2nd Edition, Springer-Verlag. [Имеется перевод: К. Йенсен, Н. Вирт. Паскаль. Руководство для пользователей и описание языка. М.: Финансы и статистика. 1982.]
Коттам (Cottam I. D.)
(1980) Functional Specification of the Modula Compiler: Release 3, Unlversiti of York Computer Science Report, YCS 33, England.

Коул (Cole A. J.)
(1976) Macro Processors, Cambridge Univ. Press.
Коулман, Хьюс, Пауэлл (Coleman D., Hughes J. W., Powell M. S.)
(1979) Engineering Data Processing Programs for Multiple Microprocessors, Microprocessors and Microsystems, **3**, **2**, p. 83—86.
Кнут (Knuth D. E.)
(1968) Fundamental Algorithms, in The Art of Computer Programming, 1, Addison Wesley. [Имеется перевод: Д. Кнут. Искусство программирования для ЭВМ. — М.: Мир, 1976.]
Кэмпбелл, Хаберман (Campbell R. H., Habermann A. N.)
(1974) The Specification of Process Synchronisation by Path Expressions, in Lecture Notes in Computer Science, **16**, Eds. Qoos and Hartmanis, p. 89—102, Springer-Verlag.
Лампсон (Lampson B. W., et al.)
(1977) Report on the Programming Language Euclid, ASM Sigplan Notices, **12**, **2**.
Лисков, Снайдер, Аткинсон, Шafферт (Liskov B., Snyder A., Atkinson R., Schaffert C.)
(1977) Abstraction Mechanisms in CLU, Comm ACM, **20, 8**, p. 564—576.
Нестор (Nestor J.)
(1978) Redl: Language for Ironman Requirements, Intermetrics Inc., USA.
(1978) Reference Manual for the Redl Language, Intermetrics Inc., USA.
Нолан, Хокинс, Пайл, Ванд (Nolan C. J., Hawkins N. P., Pyle I. C, Wand I. C.)
(1979) Modula on the Intel 8080 Microprocessor, University of York Computer Science Report, YCS 26, England.
Пайл (Pyle I. C.)
(1979) Input/Output in High Level Programming Languages, Software P and E, **9**, p. 907—914.
(1980) Axioms for User-Defined Operators, Software P and E, **10**, p. 307— 318.
Рандел (Randell B.)
(1975) System Structure for Software Fault Tolerance, IEEE Trans. Soft Eng., **1, 2**, p. 220—232.
Ратнер, Латтен (Rattner J., Lattin W.)
(1981) Ada determines architecture of 32 Bit Microprocessor, Electronics,, p. 119—126.
Рихардс (Richards M.)
(1969) BCPL: A Tool for Compiler Writing and Systems Programming,, AFIPS Conf. Proc., Spring Joint Computer Conference, **34**, p. 557.
Рол (Rohl J. S.)
(1975) An Introduction to Compiler Writing, MacDonald and Jane, Computer Monograph Series.
Рунсиман (Runciman C.)
(1980) Modula and a Vision Laboratory, University of York Computer Science Report, YCS 34, England.
Симпсон, Джексон (Simpson H. R., Jackson K.)
(1979) Process Synchronisation in Mascot, Computer J., **22, 4**, p. 332—345.
Смедена, Хейманс, Лоренцини (Smedena C. H., Heymans F., Lorenzini D.)
(1979) The Definition of PL Modula, Philips Laboratories, Technical Note
1593, New York.
Тенант (Tennant R. D.)
(1978) Another Look at Type Compatability in Pascal, Software P and E. **8**, p. 429—437.

Уэлс, Мак-Киг (Welse J., McKeag M.)
(1980) Structured System Programming, Prentice-Hall.
Уэлш, Бустард (Welsh J., Bustard D.)
(1979) Pascal Plus: Another Language for Modular Multiprogramming, Software P and E, **9, 11**, p. 947—958.
Уэлш, Элдер (Welsh J., Elder J.)
(1979) Introduction to Pascal, Prentice-Hall.
Уэлш, Листер (Welsh J., Lister A.)
(1981) A Comparative Study of Task Communication in Ada, Software P and E, **11, 3**, p. 237—250.
Уэлш, Снеерингер, Хоар (Welsh J., Sneeringer W. J., Hoare C. A. R.)
(1977) Ambiguities and Insecurities in Pascal, Software P and E, **7**, p. 685—696.
Фрогат (Froggat T. J.)
(1978) Fractions versus Fixed-Point, University of York, Internal Report. Хаберман (Habermann A. N.)
(1973) Critical comments on the programming language Pascal, Acta Informatica, **3**, p. 47—57.
Хаберман, Насси (Habermann A. N., Nassi I. R.)
(1980) Efficient Implementation of Ada Tasks, Computer Science Report, CMU-CS-80-103, Carnegie Melon University.
Хансен (Hansen P. Brinch)
(1973) Operating System Principles, Prentice Hall, N. J.
(1975a) The Programming Language Concurrent Pascal, IEEE Trans Soft. Eng., **1, 2**, p. 199—207.
(1975b) A Real Time Sheduler, Information Science Report, California Institute of Technology.
(1978) Distributed Processes: A concurrent programming concept, Comm ACM, **21, 11**, p. 934—941.
Хант (Hunt J. G.)
(1980) Interrupts, Software P and E, **10**, p. 523—530.
Хиббард, Хисген, Розенберг, Шерман (Hibbard P., Hisgen A., Rosenberg J., Sherman M.)
(1980) Programming in Ada: Examples, Report CMU/CS/80/149, Carnegie Melon University.
Холден, Ванд (Holden J., Wand I. C.)
(1977) Experience with the Programming Language Modula, Proc. of IFAC/IFIP Real Time Programming Language Seminar, Eindhoven, Ed. K. Smedema, Pergamon Press.
(1980) An assessment of Modula, Software P and E, **10, 8**, p. 593—622.
Хоар (Hoare C. A. R.)
(1972) Notes on Data Structuring, in Structured Programming, Academic Press. [Имеется перевод: К. Хоор. О структурной организации данных. В кн.: У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование - М.: Мир, 1978.]
(1974) Monitors: An operating System Structuring Concept, Comm ACM **17, 10**, p. 549—557.
(1976) Communicating Sequential Processes, Comm ACM, 21, 8, 666 - 677.
Цан (Zahn C. T.)
(1974) A Control Statement for Natural Top-Down Structured Programming, in Lecture Notes in Computer Science, 19, p. 170—180, Ed. Robinet, Springer-Verlag.

Шоу (Shaw C. J.)
(1963) A Specification of Jovial, Comm ACM, **6, 12***,* p. 721—735.
Шоу, Вульф, Лондон (Shaw M., Wulf W. A., London R. L.)
(1977) Abstraction and Verification in Alphard, Comm ACM, **20, 8**, p. 553 - 564.
Эндрюс (Andrews G. R.)
(1979) The Design of a Message Switching System: an application and evaluation of Modula, IEEE Trans. Soft. Eng., **5**, p. 138.
Янг (Young S. J.)
(1979) SYCOL Informal Language Specification, Control Systems Centre Report No, 448, UMIST, Manchester.
(1980) Low Level Device Programming from a High Level Language, IEE Proc PtE, **127**, p. 37—44.
(1981) Improving the Structure of Large Pascal Programs, Software P and E, **11, 9**, p. 913—927.
BSI (1979) Draft Standard RTL/2, British Standards Institute, DPS/13 paper 79/63337.
CCITT (1980) CHILL Language Definition.
DARPA (1980) «STONEMAN», Defense Advanced Research Projects Agency, Arlington, Virginia.
DEC (1972) POP 11/40 Processor Handbook, Digital Equipment Corporation, Mass.
IBM (1971) PL/1 Checkout and Optimising Compilers: Language Reference Manual, IBM Corporation, Number SC33-009-1.
ICI (1973) MTS Operating System (POP 11), ICI Mond Division, Runcorn, England.
Intel (1977) SBC 8020 Hardware Reference Manual, Intel Corporation.
Softech (1978) Blue: Language for Ironman Requirements, Softech, USA.
SPL (1974) RTL/2 Language Specification, SPL International, England.
SRI (1978) Yellow —a Language designed to the Department of Defense
Ironman Requirement, Stanford Research Institute, USA.

ИМЕННОЙ УКАЗАТЕЛЬ

Аткинсон (Atkinson R.) 388 Ахо (Aho A. V.) 67
Барнес (Barnes J. G. P.) 59, 92, 129, 176, 224, 253, 254, 386 Бастард (Bustard D.) 146, 176, 389 Брандес (Brandes J.) 176, 386 Брон (Bron C.) 225, 386
Ванд (Wand I. C.) 293, 315, 316, 386, 388, 389
Везеролл (Wetherall P. R.) 386
Вейнгаарден (Van Wijngaarden A.) 59, 130, 250, 386 Вирт (Wirth N.) 22, 94, 129, 145, 176, 199, 225, 252, 283, 310, 316, 386, 387
Вихман (Wichmann B. A.) 45, 386
Вудвард (Woodward P. M.) 59, 92, 176, 198, 224, 386 Вульф (Wulf W. A.) 145, 390
Гешке (Geschke C. M.) 60, 93, 225, 253, 386, 387 Горман (Gorman B.) 386 Грис (Gries D.) 67, 387
Грогоно (Grogono P.) 22 Гудинаф (Goodenough J. B.) 387
Даглес (Dagless E. L.) 177, 387 Дал (Dahl O.-J.) 145, 252, 387 Даусинг (Dowsing R.) 177, 387 Даусон (Dowson M.) 160, 387 Дейкстра (Dijkstra Ё. W.) 102, 149, 387
Джексон (Jackson K.) 176, 388
Джемисон (Jamieson M. J.) 45, 387
Ихбиа (Ichbiah J. D.) 60, 93, 176, 225, 253, 320, 387
Иенсен (Jensen K.) 22, 387
Кнут (Knuth D. E.) 90, 388
Коул (Cole A. J.) 388
Коулман (Coleman D.) 147, 388
Коттам (Cottam I. D.) 283, 387
Кусо (Cousot P.) 387
Кэмпбелл (Campbell R. H.) 177, 388
Лампсон (Lampson B. W.) 60, 253, 388
Латтен (Lattin W.) 244, 388 Л исков (Liskov B.) 145, 252, 388 Листер (Lister J.) 389 Лондон (London R. L.) 145, 390 Лоренцини (Lorenzini D.) 388
Мак-Киг (McKeag M.) 131, 389
Морис (Morris J. H., Jr.) 60, 386
Мюрхауг (Myhrhaug B.) 387
Насси (Nassi I. R.) 175, 389
Нестор (Nestor J.) 60, 93, 130, 176,, 388
Нолан (Nolan C. J.) 283, 388
Нюгорд (Nygaard K.) 387
Пайл (Pyle I. C.) 127, 388 Пауэлл (Powell M. S.) 388
Рандел (Randell B.) 217, 388 Ратнер (Rattner J.) 244, 388 Риссен (Rissen J. P.) 387 Рихарде (Richards M.) 248, 388 Розенберг (Rosenberg J.) 389 Рол (Rohl J. S.) 118, 388
Рунсиман (Runciman C.) 315, 388
Сатертвайте (Sattcrthwaite E. H.) 60,, 225, 253, 386, 387 Снеерингер (Sneeringer W. J.) 60, 389
Симпсон (Simpson H. R.) 176, 388 Смедена (Smedena C. H.) 283, 388 Снайдер (Snyder A.) 388
Тенант (Tennant R. D.) 60, 388
Ульман (Ullman J. D.) 67 Уэлс (Welse J.) 389 Уэлш (Welsh J.) 22, 131, 146, 176г 389
Ферран (Ferran G.) 253, 387
Фоккинга (Fokkinga M. M.) 225, 386 Фрогат (Froggat T. J.) 61, 389
Хаберман (Habermann A N.) 60, 175, 177, 181, 388, 389 Хансен (Hansen P. Brinch) 145, 157, 161, 176, 252, 389 Хант (Hunt J. G.) 389 Хейманс (Heymans F.) 388
Хелиард (Heliard J. C.) 387
Хиббард (Hibbard P.) 389 •Хисген (Hisgen A.) 389 Хоар (Hoare C. A. R.) 60, 62, 157,. 161, 176, 389
Хокинс (Hawkins N. P.) 388 Холден (Holden J.) 293, 315, 389 Хьюс (Hughes J. W.) 388
Цан (Zahn C. T.) 103, 389
Шафферт (Schaffert C.) 388
Шерман (Sherman M.) 389 Шоу (Shaw C. J.) 60, 92, 145, 224, 252, 390
Эйхентопф (Eichentopf S.) 386
Элдер (Elder J.) 22, 389 Эндрюс (Andrews G. R.) 315, 390
Янг (Young S. J.) 61, 144, 199, 390

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Абстрактные типы данных 131, 339
Аналого-цифровые преобразователи
53, 376
Архитектура фон Неймана 19
Атрибут (attribut) 28
Библиотечные пакеты в языке АДА
322, 352
Битовый тип (bits type) 93, 283, 369
Блоки 105
— в языке Ада 336
— область действия имени 106
— реализация 107
Блочная структура 107, 108
— — в RTL/2 262
Брусок данных (data brick) в RTL/2
263
— — **svc** 267
— процедурный (procedure brick) 263
— стековый (stack brick) 263
Ввод-вывод (input/output) абстрактная модель 187
— адресация регистров 181
— буферизованный 194
— — в вычислительных системах общего назначения 177
— — — реального времени 179
— — — языке Ада 347, 348, 354
— — — Модула 293, 294, 302
— — — RTL/2 267, 268
— высокого уровня 241
— — описание регистров 183
— передача блока данных с использованием МПДП 195, 196
Взаимно исключающий доступ (mutual exclusion) 149, 190
— — в языке Модула 314
«Внуки» процессов (grandchild) 154
Время абстрактный тип 132, 300
— ответа системы 11
Входы в языке Ада 343
Вынужденные преобразования в RTL/2 256
Выражения для путей доступа (path expressions) 177
Границы массива 64, 92
Двоичные числа 51
Двойственность именования 114
Двусмысленность описаний 34, 85
Деление целых 44
Динамические структуры данных (dynamic data structures) 78
— — преимущества 80, 82
— — пример 79, 80
— проектирование 82, 83
— — распределение памяти 79
— — реализация 89, 90
— — список 78, 79, 85
— — типизация 83, 84
— — уничтожение 83, 87, 88, 334
Динамическое создание задач 346
Директива safe в языке Red 130
Дискриминанты 332, 342
Дробные типы (fraction types) 52, 53
— в языке RTL/2 256, 257
— двойная длина 53, 257
— недостатки 55
— преимущества 55
— пример использования 53
Завершение по времени (timeout) 170
Задача поставщик/потребитель (producer/consumer) 148
— — — решение при помощи монитора 157
— — — — — — семафора 150, 151
— — —— с применением модели рандеву 166
Задачи [*см. также* процессы] в операционных системах 147
— управлении процессами 10, 11
— языке Ада 343
—— RTL/2 269
Задержка процесса (delay process) 171, 346
— оператор 346
Записи (records) 68
— вариантные 69, 70
— в языке Ада 332
— надежность 71
— подтипы 69
— реализация 75
— тип 68
— — фиксированный 68, 69
— объединения 72, 73
— в языке Ада 332
— — — Модула 285
— — — — RTL/2 260
Значения по умолчанию 339
Имя со спецификатором **pervasive** 118
Индексирование массивов 62, 63
Инициализация переменных 240
Интерфейс с оператором 12, 306
Интерфейсные модули 293, 294, 314
Исключения (exeptions) в процессах 219, 351
— в языке Ада 349
— возбуждение 204
— описание 208, 209

— *отметить* 211, 212
— правила распространения 213, 351
— *разобраться* 211, 212
— реализация 223
— *уйти* 211, 212
Квалификаторы (qualifiers) 37, 325
Класс параметров in 112, 113
— — inout 112, 113
— — out 112, ИЗ
— типы 145
— Конструктор записи 71
— множества 76
Контекстное ' переключение (context switching) 271
Критерии проектирования, гибкость 16
— — мобильность 16, 17
— — надежность 13, 14
— — простота 16
— — удобочитаемость 14, 15
— — эффективность 17
Куча (heap storage) 90
Логический тип 37
— в языке Ада 326
— — операции 38
реализация 38
Мантисса 46
Массив (аггеу) в языке Ада 330
— — Модула 284
— — RTL/2 259
— вырезки (slices) 65
— динамический 66
— индексы 64
— многомерный 66
— неограниченный 330
— конструктор 67
— в языке Ада 329
— подтипы 65
— понятие 62, 63
— реализация 67, 68
— тип 62, 66
— эквивалентность 65
Метки (labels) 203
Механизм прямого доступа 187, 195,196
Множества 72
— операции 76
— представление на машине 184
— реализация 77
Модули ввода-вывода 185, 186, 190, 291
Модуль (module) 133
— в языке Модула 287, 288
— использование 135, 136
— механизмы машинного оборудования 186, 187, 189, 190, 290
— правила области действия 141
— раздельная компиляция 238, 239
— разработка 136, 137
— реализация 144
Мониторы 156, 157, 190, 220, 289
Набор символов ASCII 39
Наследование атрибутов типа 28
— — — в языке Ада 324
— операций 29
Область действия имени (scope of name) в блоках 105
— — — — модулях 141
— — — — процедурах 116
— — — правила 106
— — спецификатор **pervasive** 118
— ошибки 206
Ограничения диапазона 30
Оператор *выпвв* (*doio*) 291, 292, 313, 317
— окончания цикла (loop termination) 103
— *освободить* (**release**) 149
реализация 278, 290
— *отключить* (*fail*) 220
— *прекратить* (*abort*) 155, 169
— приема (accept) 162, 343
— управления (control) 95
— в языке Паскаль 96
— разработка 96
— цикла (loop) в языке Ада 336
— — — — Модула 286
— — — — RTL/2 261
— — определения 101
— **case** 100, 101
— — в языке Ада 335
— — Модула 285
— **exit** в языке Ада 336
— — — Модула 285, 286
------ разработка 102
— **for** в языке Ада 336, 337
— — — Паскаль 97, 247
— — — RTL/2 261
— — разработка 101
— — **goto** 103, 202
— — в языке Ада 336
— — — RTL/2 263
— **if** 100
— — в языке Ада 335
— — — Модула 285, 286
— — — Паскаль 97
— — — RTL/2 261
— **init** 155, 156
— **raise** 209
— в языке Ада 350
— repeat в языке Модула 285
Паскаль 96

— **return** 121, 265, 338
— **select** 164, 343, 346
— **switch** 263
— val 259
— [префикс] **while** 101
— — в языке Ада 336
— — Модула 285
— — Паскаль 96
— — RTL/2 262
— **with** 285
Операторы отношения 37, 329
Операции 126
— в языке Ада 328, 329
— — Модула 285
— — RTL/2 256, 257, 258
— совмещение (overloading) 127
— свойства 127, 128, 229
— сдвига (shift) 258
Операция ждать (wait) 149
— в языке Модула 289, 292, 311, 316
— — реализация 278
— — послать (send) 149
----- в языке Модула 289, 292, 311, 316
— — реализация 278
— **and** 37, 258, 285, 327
— **div** 285
— **mod** 44, 285, 328, 329
— **not** 37, 258, 285
— **or** 37, 258, 285
— **rem** 328 Описания mode 260
Описатель процесса (process descriptor) в языке Модула 310
Определение диапазона (range check) 31
Оптимизация вызовов процедур 120
— — модели рандеву 174
Ошибки в программах 20, 200
— восстановление 215, 216
— — в языке Ада 349
— — RTL/2 268
— обнаружение 199, 200
— обработка 199, 200, 201
Пакеты (packages) 360
Параметр типа пространство (space) 198, 199
Перегруженность или совмещение (overloading) перечислимого типа 37
— — в языке Ада 325, 339
— — операции 29, 127
— — подпрограммы 122
Перечислимые типы (enumerated types) 34
— в языке Ада 325
— — операции 34, 35, 326
— определенные пользователем 183
—отображение на машинные регистры 183, 347
— — перегруженные 37
— — проблемы 36
Подкомпонента (subunit) 354
Подтипы (subtypes) 32, 33, 47, 65, 70, 84, 228, 323, 329, 333
Порождающие программные единицы 241, 242, 343
Порожденные (child) процессы 154
Порядковое значение перечисления 39
Постепенное уточнение 131
— — данных 61
— — программ 94
Предложение **use** 341
— with в языке Ада 323
Преобразование типов 23
Прерывания 187
— — в языке Ада 349
— — Модула 313
— с помощью сигнала 190
— управление 187, 292
Приватная часть спецификации модуля 133
— пакета 341
Приватные типы 341, 342
— — **limited private** 342
Признаки статуса (status flags) 38, 186
— — обновление (polling) 186
Программирование последних целей (last wishes) 215, 218
Прохождение сообщений 167, 168
Процедура *оградить* (secure) 149
— — реализация 278
— *освободить* (release) 149
— — реализация 278
— переименования (renamingrs) 345, 363
— распределения памяти (allocation) 78, 87
Процедуры (procedures) 94, 108
— абстрактные классы параметров 11, 112
— в языке Ада 337
— — Модула 286
— — RTL/2 264
— передача параметра 113
— понятие 108
— реализация 118, 119
— типы параметров ПО, 111
—управление именной областью (name scope control) 118
Процессы 11, 12, 153

— в языке Модула 289
— исключения 218, 219
— описание 153
— параметры 154
— правила завершения 104
— прекращение 155, 169
— реализация 172
Прямое цифровое управление ПЦУ 11
Разделение времени (time sharing) 173 Раздельная компиляция 238
— — в языке Ада 352
— — — — RTL/2 266
Разгрузка данных 10 Рандеву (rendezvous) 161
— асимметричное 162
— модель 161, 164
— недетерминированное 164, 165
— предохранитель 165
— симметричное 163
Расширение типа (type extension) 24
Расшифровка ссылок (dereferencing) 86, 87, 259, 334
Родительский (parent) процесс 154
Реальное время (real time) определение 9, 10
— — требования к языку 17, 18
— — характеристики 11, 12
Сборка мусора (garbage collection) 87
Семафор 149, 290, 346 Сигнал (sygnal) 149 Символьные типы (character types) 39
— — в языке Ада 326
— — операторы 39
— — перегруженные 40, 42
Синхронизация двух процессов 150
— операций ввода-вывода 189
— при использовании мониторов 160
— помощи рандеву 161 Система управления процессами 10
Системы со встроенной вычислительной машиной (embedded computer systems) 10
Составление очереди задач в реальном времени 293
— расписания (sheduling) 274, 295, 312
Спецификации в языке Ада 358
Спецификация интерфейса 139
Список использования [use] (use list) в модулях 136, 139, 143
— — — процедурах 117, 286
— — — языке Модула 287
— (define) в модулях 133, 137, 139, 143
— — — языке Модула 288
Ссылочный тип (reference type) в языке RTL/2 259
Ссылочные (access) типы в языке Ада 334
Статические переменные 78
Стек (stack) использование в процедуре вызова 119
— реализация 265, 266, 288, 340
Строка (string) битов 77
— данных в языке Ада 330
Структурное программирование 18
Структурирование программ 94
Тестирование и отладка программ 20, 135, 309
Тип 17, 18
— абстрактный 131, 339
— анонимный (anonymous) 28, 33
— байтовый 256
— битовый 93, 283, 369
— динамические данные 78, 79
— аадач (task) 346
— записи (records) 69
— логический (Boolean) 37
— массива (array) 62, 63
— множественный (set) 75
— объединение (union) 73
— проверка границ (range checking) 30
— переименование (renaming) 32
— перечислимый (enumerated) 34, 35
— плавающий (floating point) 46
— производный (derived) 25
— простой 19
— символьный (character) 40
— указательный (pointer) 78
— фиксированный (fixed point) 48
— целый (integer) 43
— числовой (numeric) 42
Типизация сильная (strong typing) 24
— слабая (weak) 22 Точность спецификация 46, 56
Удаленная система приема данных 355
Указатели (pointer) висячие (dangling) 87, 88
— в языке Ада 334
— использование 79, 80
— расшифровка [обратная операция к получению ссылки] (dereferencing) 79, 86
Условия 38

Фиксированные типы 48
— — в языке Ада 327
— приближенное представление 50, 51
— пример использования 57
— точное представление 50
— требования 49
Фильтр нижних частот по Чебышеву 53, 57
Функции 94
— в языке Ада 337
— побочные эффекты 108, 122
— совмещение (overloading) 122, 123
— чистые 122
Функция предшествования (predecessor) 35, 326
— *пор* 39
— следования (successor) 35, 326
Целые типы 43
— — в языке Ада 326
— — — Модула 284
— — — RTL/2 256
— диапазон значений 43
— мобильность 44
— операции 44
Часы реального времени 11, 291, 299
Эквивалентность именная 27, 228
— структурная 27, 228
— типов 27, 228
Экспонента 46
Язык Ада 59, 92, 129, 146, 176, 199, 225, 226, 250, 253, 328—385
— Алгол 60 59, 225
— Алгол 68 59, 61, 130, 250, 251
— Альфард [Alphard] 145
— Меза (MESA) 60, 93, 225, 253
— Модула 129, 145, 175, 199, 225, 250, 252, 282—318
— Параллельный Паскаль 145
— Паскаль 22, 60
— Паскаль Плюс 146, 176
— Симула 145, 252
— CHILL 320
— CLU 145, 252
— CORAL 66 59, 60, 92, 176, 198, 224
— CSP 176
— JOVIAL 60, 92, 176, 224
— 215 93, 253
— PEARL 92, 176
— PL/1 225
— RTL/2 59, 92, 129, 175, 224, 250, 251, 253—282
— SYCOL 61
Языки Айронман [«особо надежные языки»] (Ironman) 60, 176, 199, 319
— Blue [голубой] 60, 176
— Green [зеленый] 60, 93, 176, 225
— Red [красный] 60, 93, 130, 176
— Yellow [желтый] 60, 176, 199, 225

ОГЛАВЛЕНИЕ

От редактора перевода 5
Предисловие 7

Часть I. **ПРОЕКТИРОВАНИЕ**

Глава **1. Требования к проектированию языков реального времени . 9**
1 . 1 . Характеристики систем реального времени 9
1.2. Общие критерии проектирования 13
1.2.1. Надежность 13
1.2.2. Удобочитаемость 14
1.2.3. Гибкость 16
1.24. Простота 16
1.2.5. Мобильность 16
1.2.6. Эффективность 17
1.3. Требования к языку для систем реального времени 17

Глава 2. **Простые типы данных 19**
2.1. Основные понятия 19
2.2. Механизмы типизации 22
2.2.1. Слабая типизация 22
2.2.2. Сильная типизация 24
2.2.3. Производные типы 25
2.2.4. Эквивалентность типов 27
2.2.5. Наследование атрибутов 28
2.2.6. Ограничения 30
2.2.7. Подтипы 32
2 2.8. Анонимные типы и подтипы 33
2.3. Перечислимые типы 34
2.3.1. Определенные пользователем перечислимые типы 35
2.3.2. Логические типы 37
2.3.3. Символьные типы 39
2.4. Числовые типы 42
2.4.1. Целые типы 43
2.4.2. Плавающие типы 46
2.4.3. Фиксированные типы 48
2.5. Типизация в практических языках 59

Глава **3. Структурные типы данных** 61
3.1. Структурирование данных 61
3.2. Массивы 62

3.2.1. Понятие массива ... 62
3.2.2. Типизация с помощью массивов ... 64
3.2.3. Многомерные массивы ... 66
3.2.4. Конструкторы массивов ... 67
3.25. Реализация массивов ... 67
3.3. Записи ... 68
3.3.1. Фиксированные записи ... 68
3.3.2. Вариантные записи ... 69
33.3. Объединения ... 72
3.3.4. Варианты или объединения? ... 74
3 3.5. Реализация записей ... 75
3.4. Множества ... 75
34.1. Понятие множества ... 75
3.4.2. Реализация множеств ... 77
3.5. Структуры динамических данных ... 78
3.5.1. Статические и динамические переменные ... 78
3 5.2. Вопросы проектирования ... 82
3.5.2.1. Типизация динамических данных ... 83
3.5.2.2. Расшифровка ссылок ... 86
3.5.2.3. Создание и уничтожение динамических объектов данных ... 87
3.5.3. Реализационные аспекты ... 89
3.6. Структурные типы данных в существующих языках ... 92

Глава 4. Классические программные структуры ... 94
4.1. Структурирование программ ... 94
4.2. Операторы управления ... 95
4.2.1. Основные формы ... 95
4.2.2. Разработка операторов управления ... 96
4.2.3. Примеры конструкций ... 99
4.2.4. Ненормальное завершение операторов повторения ... 102
4.3. Блочная структура ... 103
4.31. Предпосылки ... 103
4.32. Блоки ... 105
4.3.3. Реализация блоков ... 107
4.4. Процедуры ... 108
4.4.1. Понятие процедуры ... 108
4.4.2. Параметры ... 110
4.4.3. Правила области действия ... 116
4.4.4. Аспекты реализации ... 118
4.5. Функции и операции ... 120
4.5.1. Функции ... 120
4.5.2. Совмещение ... 122
45.3. Операции ... 126
4.6. Программные структуры в практических языках ... 129

Глава 5. Механизмы абстракции ... 131
5.1. Абстрактные типы данных ... 131
5.2. Что еще связано с введением понятия модуля ... 134
5 3. Разработка механизма организации модуля ... 136
5.3.1. Общая структура ... 136
5.3.2. Спецификация интерфейса ... 139
5.3.3. Правила именной области действия ... 141
5.4. Аспекты реализации ... 144
5.5. Механизмы абстракции в практических языках ... 145

Глава 6. Параллельность ... 146
6.1. Параллельность в системах реального времени ... 146

6.2. Традиционный подход к мультипрограммированию................147
6.3. Процессы 153
6 4. Мониторы 156
6.5. Рандеву..161
6.6. Прохождение сообщений 167
6.7. Требования к языкам реального времени 169
6.8. Аспекты реализации 172
6.9. Параллельность в практических языках 175

Глава 7. Программирование ввода-вывода низкого уровня 177
7.1. Ввод-вывод в вычислительных системах 177
7.2. Доступ к машинному оборудованию 181
7.2 1. Адресация регистров ввода-вывода 181
7.2.2. Отображение определенных пользователем типов на машинные регистры 183
7.2.3. Модули ввода-вывода ...;...185
7.3. Модель для программирования ввода-вывода низкого уровня 186
7.3.1. Механизмы машинного оборудования 186
7.3.2. Абстрактная модель...187
7.3.3. Модель языка 189
7.4. Программирование ввода-вывода 192
7.4.1. Передача одного символа данных...192
7.4 2. Передача блоков данных с использованием МПДП 195
7.5. Программирование ввода-вывода низкого уровня в практических языках 198
Глава 8. Обработка ошибок 199
8.1. Основные понятия 199
8.2. Традиционные методы обработки ошибок...............................201
8.3. Обработка исключений 205
8.3.1. Структурная обработка ошибок 205
8.3.2. Базовые механизмы 208
8.3.3. Распространение исключений...213
8.3.4. Восстановление после ошибок 215
8.3.5. Исключения в параллельных процессах 219
8.3.6. Аспекты реализации 223
8.4. Обработка ошибок в практических языках 224
Глава 9. Проектирование языка 226
9.1. Типизация данных 226
9.2. Структура программ 230
9.3. Мультипрограммирование...232
9.4. Программирование устройств низкого уровня 235
9.5. Обработка ошибок 236
9.6. Другие аспекты проектирования языка 238
9.6.1. Раздельная компиляция..238
9.6.2. Инициализация переменных..240
9.6.3. Ввод-вывод высокого уровня...241
9.6 4. Порождающие программные единицы 241
9.7. Выбор свойств языка 243
9.8. Общие принципы проектирования..246

Часть II. **РАЗРАБОТКА**
Глава 10. **Разработка языков реального времени** 250
Глава 11. **Язык RTL/2** 253
11.1. Предпосылки 253
11.2. Язык 254

11.2.1. Формальный язык……………………………………...255
11.2.2. Стандартный ввод-вывод и восстановление после ошибок…………………………………………………………267
11.3. Использование языка RTL/2………………………………269
11.4. Обсуждение…………………………………………………279

Глава 12. Модула………………………………………………………..282
12.1. Предпосылки…………………………………………………282
12.2. Язык .... ……………………………………………283
12.2.1.Особенности последовательного программирования..283
12.2.2.Особенности параллельной обработки………………..289
12.3. Использование Модулы………………………………………..293
123.1. Составление очереди задач в реальном времению….. 293
123.2. Составление расписания для задач пользователя…….295
12.3.3.Слежение за временем…………………………………298
12.3.4.Терминальный ввод-вывод…………………………….302
12.3.5.Интерфейс с оператором………………………………306
12.3.6.Программа управления задачами………………...……307
12.4. Реализация Модулы……………………………………………309
12.5. Обсуждение…………………………………………………….315

Глава 13. Ада……………………………………………………………318
13.1. Предпосылки…………………………………………………..318
13.2. Язык……………………………………………………………..321
13.2.1. Общий стиль……………………………………...321
13.2.2. Простые типы данных…………………………………323
13.2.3. Структурные типы данных……………………………329
13.2.4. Классическая программная структура………………335
132.5. Абстракция данных……………………………………339
13.2.6. Параллельность………………………………………343
13.27. Программирование ввода-вывода низкого уровня….347
13.2.8. Обработка исключений…………………………...349
13.2.9. Раздельная компиляция...............................…………352
13.2.10.Ввод-вывод.. ……354
13.3. Использование языка Ада…………………………………...355
13.3.1.Удаленная система приема данных……………………355
13.3.2.Спецификация…………………………………………..358
133.3. Реализация……………………………………………….365
13.4. Обсуждение……………………………………………………..380

Литература………………………………………………………………...386
Именной указатель………………………………………………………...391
Предметный указатель……………………………………………………392