

БИБЛИОТЕЧКА  
ПРОГРАММИСТА

А.Ф. ДЕДКОВ

**Абстрактные  
типы данных  
в языке  
АТ-Паскаль**



БИБЛИОТЕЧКА  
ПРОГРАММИСТА

---

А. Ф. ДЕДКОВ

# АБСТРАКТНЫЕ ТИПЫ ДАННЫХ В ЯЗЫКЕ АТ-ПАСКАЛЬ



МОСКВА «НАУКА»

ГЛАВНАЯ РЕДАКЦИЯ

ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

1989

ББК 22.18  
Д 26  
УДК 519.85

Серия основана в 1968 году  
В ы п у с к 58

Дедков А. Ф. Абстрактные типы данных в языке АТ-Паскаль.— М.: Наука. Гл. ред. физ.-мат. лит., 1989.— (Библиотечка программиста).— 200 с.— ISBN 5-02-013968-8

Практическое введение в два перспективных направления современного программирования: технику программирования с использованием концепции абстрактных типов данных и распределенное программирование в сетях ЭВМ. Изложение ведется на основе языка АТ-Паскаль, представляющего собой расширение языка Паскаль средствами работы с абстрактными типами данных и средствами взаимодействия распределенных процессов, и иллюстрируется рядом примеров. Кратко рассмотрены также аналогичные языковые средства языков Ада и Модула-2.

Для научных работников, инженеров, может быть полезна студентам.

Ил. 33. Библиогр. 35 назв.

Рецензенты:

кандидаты физико-математических наук *В. Г. Бацла*  
и *В. О. Сафонов*

Д  $\frac{1404000000-072}{053(02)-89}$  133-89

ISBN 5-02-013968-8

© Издательство «Наука».  
Главная редакция  
физико-математической  
литературы, 1989

## ОГЛАВЛЕНИЕ

Предисловие автора . . . . .	5
<b>Глава 1. Введение . . . . .</b>	<b>7</b>
1.1. Назначение книги . . . . .	7
1.2. Об абстрактных типах данных . . . . .	8
1.3. Что дает применение абстрактных типов данных . . . . .	10
1.4. О распределенном программировании . . . . .	12
<b>Глава 2. Типы данных и операции над ними . . . . .</b>	<b>13</b>
2.1. Конструирование типов в языке Паскаль . . . . .	13
2.2. Конструкторы типов как функции над типами . . . . .	18
2.3. Деструкторы типов и функции-константы . . . . .	22
2.4. Ссылочная идентичность типов . . . . .	27
2.5. Примеры применения деструкторов типов . . . . .	30
<b>Глава 3. Пакеты . . . . .</b>	<b>36</b>
3.1. Понятие пакета . . . . .	36
3.2. Присоединение и вложение пакетов . . . . .	37
3.3. Доступ к объектам вложенного пакета . . . . .	40
3.4. Ключевые параметры пакетов . . . . .	42
3.5. Позиционные параметры пакетов . . . . .	44
3.6. Пакеты-функции . . . . .	47
3.7. Процедуры с переменным числом параметров . . . . .	49
3.8. Локальные объекты и приватные типы . . . . .	51
3.9. Ограничение видимости . . . . .	53
3.10. Пакеты как процедуры периода компиляции . . . . .	53
<b>Глава 4. Основные принципы АТД-программирования . . . . .</b>	<b>56</b>
4.1. Классификация пакетов . . . . .	56
4.2. Основные свойства АТД-программирования . . . . .	57
4.3. Об эффективности АТД-программирования . . . . .	61
4.4. Способы объявления объектов АТД . . . . .	63
<b>Глава 5. Примеры пакетов . . . . .</b>	<b>67</b>
5.1. Простые пакеты . . . . .	67
5.2. Пакеты для обработки строк переменной длины . . . . .	70
5.3. Пакет для работы с комплексными числами . . . . .	81
5.4. Обработка списков . . . . .	84

5.5. Пример АТД-программы . . . . .	91
5.6. Полиморфная процедура быстрой сортировки . . .	94
5.7. Пример построения программы методом пошаговой реорганизации . . . . .	102
<b>Глава 6. Введение в распределенное программирование</b>	<b>119</b>
6.1. Основные понятия распределенного программиро- вания . . . . .	119
6.2. Средства РП-программирования языка АТ-Паскаль	126
6.3. Пакеты для РП-программирования . . . . .	134
6.4. Примеры процессов для управления ресурсами	149
<b>Глава 7. АТ-Паскаль, Ада и Модула-2 . . . . .</b>	<b>161</b>
7.1. Средства АТД-программирования языка Ада . . .	161
7.2. Средства взаимодействия процессов языка Ада	168
7.3. Модули языка Модула-2 . . . . .	173
<b>Приложение. Формальное описание языка АТ-Паскаль</b>	<b>177</b>
Список литературы . . . . .	197
Список дополнительной литературы . . . . .	198

## ПРЕДИСЛОВИЕ АВТОРА

Эта книга представляет собой краткое практическое введение в два перспективных направления современного программирования: абстрактные типы данных и распределенное программирование. В настоящее время они остаются в большой степени объектами теоретических исследований, так как большинство современных широко применяемых языков программирования не имеют адекватных языковых средств. Поэтому представляется целесообразным познакомить с этими новыми направлениями широкие круги практикующих программистов. Книга основана на результатах, полученных автором в 1983—1985 гг. при выполнении работы, связанной с созданием языковых средств распределенного программирования в сетях ЭВМ. В это время автором был разработан и реализован на ЕС ЭВМ язык АТ-Паскаль, представляющий собой расширение языка Паскаль средствами, ориентированными на использование указанных методологий программирования. Язык АТ-Паскаль представляется удобным для предварительного знакомства читателя с основами программирования с использованием абстрактных типов данных и распределенного программирования. Автор надеется, что приведенные в книге сведения помогут читателям-программистам в освоении и практическом применении перспективных методов программирования, независимо от того, в рамках какого именно языка они будут применяться ими в дальнейшем. Первые пять глав книги и приложение могут служить в качестве пособия при практическом применении компилятора языка АТ-Паскаль, построенного на базе широко известного в программистских кругах компилятора Паскаль-8000. К сожалению, в доступную пользователям версию компилятора АТ-Паскаль не включены средства взаимодействия сетевых процессов, экспериментальная реализация которых была проведена в рамках сетевого программного комплекса Диалог, без которого эти средства не могут использоваться.

Автор благодарен рецензенту В. О. Сафонову, внимательно прочитавшему рукопись книги и сделавшему ряд полезных замечаний, и Фидрусу В. И.— автору комплекса сетевого программного обеспечения Диалог, оказавшему неоценимую помощь в реализации средств взаимодействия процессов языка АТ-Паскаль. Автор выражает также глубокую признательность разработчикам компилятора Паскаль-8000 Джеффри Тобиасу и Гордону Коксу из Австралийской комиссии по атомной энергии, чья прекрасная разработка послужила основой компилятора языка АТ-Паскаль.

## ГЛАВА 1

### ВВЕДЕНИЕ

#### 1.1. Назначение книги

Развитие языков программирования в последние десять лет шло по нескольким более или менее независимым направлениям. Это в первую очередь программирование с использованием абстрактных типов данных, распределенное программирование, объектно-ориентированное программирование, логическое программирование. Каждое из этих направлений стремится решить тот или иной комплекс проблем, обеспечить средства для решения того или иного класса задач. Так, программирование с абстрактными типами данных и объектно-ориентированное программирование представляют собой новые подходы к структуризации программ, имеющие целью повысить качество программ (надежность, модифицируемость, познаваемость). Распределенное программирование дает новые языковые средства для построения систем распределенной обработки данных в сетях ЭВМ. Логическое программирование предназначено для решения задач, возникающих при построении систем искусственного интеллекта. В рамках каждого из направлений были созданы новые языки программирования или разработаны расширения существующих. Однако большинство из них имеют экспериментальный характер и не вошли еще в повседневную практику большинства программистов. Видимо, это и объясняет тот факт, что эти новые языковые средства не получили еще большого распространения. Ведь для хорошего программиста язык программирования — это не набор правил кодирования алгоритмов, это — стиль мышления. И любая новая идея в области программирования не воспринимается, если она не может быть выражена на привычном для программиста и доступном для него языке.

Данная книга посвящена двум из указанных выше направлений: программированию с использованием абстрактных типов данных и распределенному программированию. Объединение их вместе оправдано, так как языковые средства того и другого направления



могут весьма эффективно применяться совместно. Будучи программистом и адресуясь к программистам, автор хорошо понимает, что при изучении новых языковых средств или идей один-два примера дают гораздо большее понимание, чем несколько страниц текста. Это определило стиль изложения материала. Все новые возможности и приемы будут демонстрироваться в первую очередь на примерах. Следует предупредить, что эта книга не является учебником по программированию для начинающих. Такие учебники существуют [1, 2, 3], и повторять здесь их содержание не имеет смысла. Предполагается, что читатель является программистом, знающим по крайней мере один язык программирования высокого уровня. Наилучшим вариантом является Паскаль, так как вводимые в книге языковые средства основаны именно на нем. Можно надеяться, что большая часть изложенных идей и приемов будет понятна и без детального знакомства с языком Паскаль. Такие конструкции, как процедуры, циклы и условные операторы, понятны всем программистам, а там, где необходимо, специфические особенности языка Паскаль и особенно новые вводимые в него средства будут описываться.

Несколько слов о выборе языка для записи примеров. В качестве него выбрано разработанное и реализованное автором расширение языка Паскаль (называемое АТ-Паскаль). Язык программирования Паскаль широко распространен во всем мире и получает все большее распространение в нашей стране. В качестве альтернатив при описании средств программирования с абстрактными типами данных могли бы использоваться такие языки, как CLU [4], Alphard [5] и Ада [6]. Выбор в пользу языка АТ-Паскаль был сделан по следующим соображениям: введенные в Паскаль средства почти не уступают по мощности аналогичным средствам языка Ада; язык Паскаль проще для изучения, а введенные в него расширения невелики и просты для понимания и использования. Кроме того, в языке АТ-Паскаль имеются средства взаимодействия распределенных процессов, так что с его помощью можно продемонстрировать возможность совместного использования средств программирования с абстрактными типами данных и средств распределенного программирования. И, наконец, не последнюю роль сыграла доступность для автора компилятора языка АТ-Паскаль. Все приведенные в книге примеры проверены на ЭВМ.

## 1.2. Об абстрактных типах данных

Рассмотрим вкратце использование понятия типа в языках программирования высокого уровня. Каждая используемая в программе переменная (точнее, каждый объект, будь то константа или функция) характеризуется ее типом. Тип определяет, во-первых,

множество значений, которые могут принимать переменные, принадлежащие данному типу, и, во-вторых, совокупность операций, допустимых над ними. Например, тип INTEGER в языках Алгол, Фортран и Паскаль или тип FIXED BINARY в ПЛ/1 задает в качестве множества значений множество целых чисел, а в качестве допустимых операций — арифметические операции, операции сравнения, операцию присваивания и ряд встроенных функций. Эти свойства объектов данного типа и есть то существенное, что должен знать о них программист, т. е. *что* можно делать с переменными типа INTEGER. Именно эти свойства обычно приводятся в описании любого языка программирования. Но существует и то, что мы будем далее называть представлением или реализацией типа. Реализация определяет, *как* представляются объекты данного типа в памяти машины и *как* над ними выполняются операции. Множество доступных для использования программисту типов задается в определении языка программирования, а их реализация определяется при создании компилятора с этого языка для конкретной машины. На разных машинах реализация одного и того же типа может оказаться различной. Например, на одной деление вещественных чисел может выполняться одной машинной командой, а на другой — специальной подпрограммой, выражающей операцию деления через другие. Программист, работающий на некотором языке, может и не знать деталей реализации типов, однако широко распространено мнение, что хороший программист должен их знать и использовать это знание для построения эффективных программ. Если речь идет только о выборе того или иного типа данных для экономного расхода памяти, выборе наиболее эффективно выполняющихся операций или использовании этого знания при отладке программы, с этим утверждением можно согласиться. Однако привнесение этого знания в текст самой программы, которое часто считалось вершиной программистского мастерства, в свете современных тенденций следует считать некорректным. При этом программа, правильно решая задачу, становится некорректной в рамках правил данного языка и привязанной к конкретному компилятору и конкретной машине. Известен случай, когда отлаженная программа на языке ПЛ/1 перестала работать после перекомпиляции ее другим компилятором на той же самой машине. Оказалось, что в ней была использована информация о реализации переменных типа *метка*, а в другом компиляторе их реализация была изменена. Запрет использования знания о реализации того или иного типа является одной из основных идей в программировании с абстрактными типами данных.

В ряде языков существуют такие типы данных, информация о реализации которых, если и доступна программисту, не может быть использована с какой-либо ощутимой пользой. Примерами

служат переменные типа *файл*, *область* или *имя точки входа* в языке ПЛ/1. Такие типы данных вполне заслуживают того, чтобы именоваться абстрактными. В чем же тогда различие между ними и теми, которые называются абстрактными типами в более современных языках программирования и которым посвящена эта книга? Различие в том, что первые определены в языке программирования (или, как говорят, *предопределены*) и реализация их фиксируется компилятором, а вторые определяются программистом в процессе проектирования и написания программы. Их представление в памяти машины и реализацию операций над ними задает сам программист. Эта реализация не обязательно должна выражаться через операции над *предопределенными* типами, она может использовать операции над объектами других абстрактных типов.

### 1.3. Что дает применение абстрактных типов данных

Введение в языки программирования определяемых программистом абстрактных типов данных (АТД) является следующим закономерным шагом в процессе повышения уровня языков программирования, логически вытекающим из общих тенденций развития языков. Что же дают они рядовому практикующему программисту? Поскольку их широкое применение только еще начинается, давать всеобъемлющее описание их преимуществ, видимо, еще рано. Попробуем, однако, кратко охарактеризовать эти преимущества, надеясь, что будущее подтвердит данные здесь обещания.

1) *Структуризация*. Иерархическая структуризация данных, вводимая в программу с помощью АТД, играет примерно ту же роль, что и структуризация алгоритмов, введенная концепцией структурного программирования, хорошо известной всем программистам. Она улучшает понятность программ, облегчает их отладку и модификацию. Если структурное программирование в общем случае не дает конкретных правил разбиения сложной программы на модули (кроме общих рекомендаций относительно размеров модуля, количества его параметров и точек входа и выхода), то концепция АТД непосредственно диктует правила разбиения, вытекающие из необходимости определения операций над АТД.

2) *Надежность*. Как известно из исторического опыта программирования, чем лучше программа структурирована, тем выше ее надежность, т. е. способность правильно или по крайней мере разумно работать в непредсказуемых условиях. Можно надеяться, что применение АТД, повышая структурированность программ, повлияет и на их надежность. Кроме того, в языках с АТД обычно

вводится очень жесткий контроль типов, позволяющий обнаружить многие ошибки еще на этапе компиляции программы.

3) *Легкость модификации.* В программе, написанной с использованием АТД, большинство изменений есть, по сути, изменения реализации некоторых АТД. Поскольку реализация АТД обычно сосредоточена в одном или крайне небольшом числе модулей, такое изменение легко выполнить. Если применяется строгая дисциплина программирования и другие модули не используют знания деталей реализации АТД, эти изменения не должны повлиять на правильность работы других частей программы. Любому программисту, отлаживавшему программу, состоящую из нескольких десятков модулей, знакомо, как сложно бывает порой ввести небольшое изменение в структуру данных программы, какой цепной процесс правки модулей начинается при этом и как ошибка, внесенная в программу при модификации одного из модулей, вдруг неожиданно проявляется в нарушении работы другого. Программирование с АТД должно устранить все эти неприятности.

4) *Упрощение тестирования.* Тестирование программы в основном сводится к демонстрации того, что операции над АТД реализованы корректно. При модификации программы, как ясно из предыдущего, тестированию должны подвергаться только измененные модули, так как непредсказуемое воздействие их друг на друга сводится к минимуму.

5) *Построение программ методом пошаговой детализации.* Включение средств работы с АТД в язык программирования является, по существу, не чем иным, как отражением этой методологии в языке.

6) *Формальная верификация программ.* Интенсивные работы ведутся в области применения АТД к формальному доказательству правильности программ или верификации. В теоретических работах в понятие АТД включается еще и формальная спецификация свойств объектов данного типа, позволяющая автоматически проверить корректность реализации АТД или даже вывести реализацию из свойств объекта. В этой книге мы не будем останавливаться на этих возможностях ввиду ограниченности объема и практической ориентации ее. Однако следует заметить, что в будущем эти работы, несомненно, дадут в руки практикующего программиста мощный аппарат доказательства правильности программ без традиционного тестирования.

7) *Полиморфные процедуры.* Языковые средства, вводимые в язык для работы с АТД, оказываются пригодными и для создания так называемых полиморфных процедур, т. е. процедур, минимально зависящих от типов обрабатываемых ими данных. Такие процедуры отражают только специфику алгоритма, но не специфику типов данных, поэтому могут быть без каких бы то ни было

изменений применены к целому семейству типов данных. Это позволяет накапливать не библиотеки модулей, реализующих конкретные варианты того или иного алгоритма, а библиотеки алгоритмов, записанных в виде полиморфных процедур с максимально возможной степенью общности.

#### 1.4. О распределенном программировании

Необходимость применения языковых средств, обеспечивающих возможность использования концепции АТД (будем далее называть их средствами АТД-программирования), требует обоснования, так как эти средства, по существу, не расширяют области применения языка, а только улучшают качество программ. Средства же так называемого распределенного программирования или средства программирования распределенных процессов (будем называть их для краткости средствами РП-программирования) обеспечивают такие возможности, которые не предоставляются традиционными языками программирования. А именно, они дают программисту аппарат, позволяющий строить так называемые распределенные системы обработки данных, т. е. системы, состоящие из нескольких одновременно выполняющихся на различных машинах компонент, взаимодействующих друг с другом через сеть связи ЭВМ. Хотя средства АТД-программирования и РП-программирования в большей степени «ортогональны» и могут использоваться независимо друг от друга, их объединение в одной книге достаточно оправдано, так как наибольший эффект при построении сложных программных комплексов может быть достигнут при их совместном использовании. С одной стороны, средства АТД-программирования могут быть применены для обеспечения большего удобства использования средств РП-программирования. С другой стороны, средства РП-программирования могут служить аппаратом для реализации АТД в распределенных системах. Для тех читателей, а таких, видимо, большинство, для которых доступ к распределенным сетям ЭВМ затруднен, главы, посвященные РП-программированию, могут служить всего лишь еще одним нетривиальным примером использования концепции АТД и средств АТД-программирования. Можно также надеяться, что с появлением в будущем «промышленных» языков РП-программирования рассмотренные в этих главах простейшие элементы распределенного программирования окажутся им полезны. Изложение элементов РП-программирования ведется на основе того же языка АТ-Паскаль, что и средств АТД-программирования. Он обладает простым, но достаточно мощным аппаратом взаимодействия распределенных процессов. Использование единого языка значительно облегчает задачу демонстрации возможностей совместного использования двух описываемых классов языковых средств.

## Г Л А В А 2

# ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ

### 2.1. Конструирование типов в языке Паскаль

Почти все распространенные языки программирования, разработанные в 60-х годах, предоставляют в распоряжение пользователя набор стандартных предопределенных типов данных. Мы говорим здесь о так называемых языках процедурного программирования. Имеются и другие классы языков, например языки функционального и логического программирования, где понятие типа в современном понимании отсутствует. Язык программирования Паскаль, созданный профессором Никлаусом Виртом в начале 70-х годов, был одним из первых языков, разрешавших программисту определять новые типы данных. Это было первым и необходимым шагом на пути к АТД-программированию. В настоящее время подобные средства вводятся в любой новый язык программирования процедурного типа, например, есть они в языке Ада. Средства определения новых типов языка Паскаль являются основой предлагаемого аппарата манипулирования с АТД в языке АТ-Паскаль, поэтому на них стоит остановиться особо. Материал этого подраздела предназначен для читателей, не знакомых с языком Паскаль.

В языке Паскаль имеются четыре предопределенных простых типа данных: INTEGER (целый), REAL (вещественный), BOOLEAN (логический) и CHAR (символьный). Над переменными каждого типа определен набор допустимых операций и заданы правила записи констант каждого типа. При объявлении переменных тип указывается после списка имен переменных и отделяется от него двоеточием, например,

```
VAR X, Y : REAL;  
    Z : BOOLEAN;
```

В языке Паскаль имена типов не являются зарезервированными словами, как, например, в языке Алгол. На имена типов

распространяются все правила определения области доступности имен, применяющиеся к именам переменных, процедур и функций. Новые типы данных порождаются специальными языковыми конструкциями, которые мы будем далее называть *конструкторами типов*, так как они позволяют конструировать бесконечное множество более сложных типов из более простых. Конструктор типа можно использовать непосредственно в том месте программы, где надо указать тип переменной или функции. Можно также присвоить ему уникальное имя в разделе типов блока и использовать далее это имя вместо конструктора данного типа в различных местах программы. Хотя, как мы увидим далее, использование конструктора и имени типа — далеко не одно и то же. Присвоение имени новому типу производится конструкцией вида

TYPE имя = конструктор типа;

Рассмотрим основные конструкторы типов, используемые в языке Паскаль.

1) *Конструктор скалярного типа* непосредственно задает список значений, которые могут принимать переменные данного типа. Значения задаются идентификаторами и являются константами этого типа. Заметим, что эти имена не обозначают ничего, кроме самих себя. Примеры конструкторов скалярного типа:

(СЛАБЫЙ, УМЕРЕННЫЙ, СИЛЬНЫЙ);  
(ВКЛЮЧЕН, ВЫКЛЮЧЕН, НЕИСПРАВЕН);

Конструктор скалярного типа единственный, при применении которого не используются другие типы. Над переменными скалярных типов допускаются операции присваивания и сравнения. Например,

TYPE НАПРАВЛЕНИЕ =  
    (ВПЕРЕД, НАЗАД, ВПРАВО, ВЛЕВО);  
VAR D: НАПРАВЛЕНИЕ;

...  
FOR D := ВПЕРЕД TO ВЛЕВО DO  
    IF D < > НАЗАД THEN ...

2) *Конструктор ограниченного типа* задает подмножество значений некоторого типа, называемого *базовым типом*, в качестве которого может использоваться любой предопределенный тип, кроме вещественного, или скалярный тип. Подмножество значений ограниченного типа задается минимальным и максимальным значением элементов подмножества. Примеры конструкторов ограниченных типов:

1..20  
—10..10

'A'..'F'

ВКЛЮЧЕН..ВЫКЛЮЧЕН

Тип констант, используемых для задания ограниченного типа, неявно задает базовый тип. Над переменными ограниченного типа допускаются те же операции, что и над переменными соответствующего базового типа.

Переменные ограниченных типов используются для повышения наглядности и надежности программ, так как при компиляции и при выполнении производится проверка того, не выходит ли присваиваемое переменной значение за пределы определенного ее типом диапазона. Например, если переменные объявлены так:

VAR J:1..20;

СОСТОЯНИЕ:(ВКЛЮЧЕН..ВЫКЛЮЧЕН);

то при выполнении следующих операторов будут обнаружены ошибки

J:=21;

J:=N; (\* где N = 100 \*)

СОСТОЯНИЕ:=НЕИСПРАВЕН;

3) *Конструктор регулярного типа* порождает такой тип, что принадлежащие ему переменные есть массивы в традиционном понимании, используемом в таких языках, как Фортран, Алгол, ПЛ/1. В отличие от этих языков, где массив характеризуется размером и типом элементов, в языке Паскаль он задается двумя типами: типом индекса и типом компонент (элементы массива иногда называются в языке Паскаль «компонентами», однако мы будем использовать более привычный термин «элементы»). Размер массива, т. е. количество его элементов, неявно задается типом индекса, который может быть или скалярным, или ограниченным. Общий вид конструктора массива таков:

ARRAY [I] OF E;

где I — тип индекса, а E — тип элемента. I и E могут быть либо именами, либо конструкторами типов. Объявление массива в языке Паскаль записывается так:

VAR A: ARRAY[1..10] OF REAL;

Тип индекса может быть любым скалярным или ограниченным типом, а тип элемента может быть произвольным. Это позволяет использовать при программировании на языке Паскаль такие экзотические конструкторы типов, как, например,

ARRAY[(ЮГ,ВОСТОК,ЗАПАД)] OF  
(СЛАБЫЙ,УМЕРЕННЫЙ,СИЛЬНЫЙ);



ARRAY ['A'..'G'] OF ARRAY[BOOLEAN] OF CHAR;

В последнем примере видно, каким образом в Паскале можно объявлять многомерные массивы.  $n$ -мерный массив трактуется как одномерный массив, элементами которого являются  $(n-1)$ -мерные массивы. Имеется сокращенная форма записи конструктора типа многомерного массива, более напоминающая традиционную. При ее использовании последний пример выглядит так:

ARRAY['A'..'G',BOOLEAN] OF CHAR;

Однако следует помнить, что это не более чем сокращение от общей формы конструктора массива, состоящего из массивов. Если учитывать это, становится понятным, почему, если массив  $A$  объявлен с приведенным выше типом, то допускается следующее его использование:

$A$  — обозначает весь массив;

$A['B']$  — обозначает элемент массива, имеющий тип

ARRAY[BOOLEAN] OF CHAR;

$A['B', \text{FALSE}]$  — обозначает элемент массива  $A['B']$ , имеющий тип CHAR.

Такое изящное определение понятия *тип массива* порождает, заметим, одну особенность, на которую часто указывают как на недостаток языка Паскаль. Размер массива в нем фиксируется на этапе компиляции, так как он определяется типом индекса, а количество значений данного типа не может меняться при выполнении программы. Это, конечно, неудобно для тех, кто привык работать, например, на ПЛ/1. Однако это оправдывается простотой и единообразием подхода к конструированию типов в Паскале. Далее будет показано, как дополнительные средства языка АТ-Паскаль частично компенсируют этот недостаток. Заметим также, что международный стандарт языка Паскаль [7] предусматривает, что параметры процедур и функций могут быть так называемыми конформантными (conformant) массивами. Такому формальному параметру при вызове процедуры может быть сопоставлен массив любого размера, но с тем же базовым типом индекса. Это облегчает построение процедур для работы с массивами. К сожалению, большинство имеющихся на разных машинах компиляторов языка Паскаль (в том числе и компилятор АТ-Паскаль) не реализуют эту возможность. Даже в описании стандарта предусмотрено, что она не является обязательной.

4) *Конструктор комбинированного типа* позволяет объявлять объекты, называемые *записями*, которые состоят из множества подобъектов (полей) произвольных типов. Каждое поле имеет имя,

уникальное в пределах данной записи. Операции допускаются как над отдельными полями, так и над всей записью. Записи языка Паскаль являются аналогами *структур* языка ПЛ/1. Пример конструктора комбинированного типа:

```
RECORD
    A:INTEGER;
    B:CHAR;
    C:RECORD
        D:REAL;
        E:ARRAY[1..4] OF BOOLEAN;
    END
END
```

Записи, объявленные с помощью типа, созданного таким конструктором, имеют три поля А, В и С соответственно целого, символьного и комбинированного типов. Поле С в свою очередь состоит из двух полей: поля D вещественного типа и поля Е регулярного типа. На этом примере видно, как в один конструктор типа могут быть вложены другие конструкторы. Если переменная R имеет такой тип, то доступ к ее полям обеспечивается уточненными именами вида

R.A, R.B, R.C, R.C.D, R.C.E, R.C.E[K].

5) *Конструктор множественного типа* имеет вид

SET OF T,

где T может быть скалярным, ограниченным или символьным типом. Тип T называется *базовым типом множества*. Значениями переменных множественного типа являются множества значений базового типа. Над множествами определены операции объединения (+), пересечения (\*) и вычитания (—). Допускается проверка того, принадлежит ли данное значение X базового типа множеству S. Для этого применяется конструкция

X IN S,

имеющая логическое значение. Примеры:

```
VAR R,S:SET OF 1..10;
    B:BOOLEAN;
```

```
S := [1,2,3];
```

```
R := [3,4,5];
```

```
R := S*R;  — R ПОЛУЧИТ ЗНАЧЕНИЕ [3]
```

```
S := S—R;  — S ПОЛУЧИТ ЗНАЧЕНИЕ [1,2]
```

```
B := 3 IN S; — В ПОЛУЧИТ ЗНАЧЕНИЕ FALSE
```

Примечание. В языке АТ-Паскаль два символа минус (— —) начинают комментарий, который продолжается до конца

текущей строки текста программы. Наряду с этим может использоваться и стандартная форма комментария, принятая в языке Паскаль — (\* . . . . . \*).

6) *Конструктор файлового типа* имеет вид  
FILE OF T.

Переменные файловых типов представляют собой последовательности компонент типа T произвольной длины. Эти переменные предназначены для манипулирования файлами во внешней памяти с помощью специальных встроенных процедур ввода-вывода.

7) *Конструктор упакованного типа* имеет вид  
PACKED T,

где T — конструктор типа. Переменные упакованных типов представляются в памяти машины максимально эффективно.

8) *Конструктор типа указателя* предназначен для объявления переменных, значениями которых являются адреса динамически размещаемых в памяти переменных базового типа. Конструкторы типа указателя есть

@ T (@ заменяет в AT-Паскале вертикальную стрелку)

или

REF(C) (только в AT-Паскале).

Здесь T — обязательно имя базового типа (имя, а не произвольный конструктор), а C — произвольный конструктор типа. Например, значением переменной

VAR R:REF(ARRAY[1..4] OF 1..10);

будет адрес массива из четырех элементов типа 1..10.

## 2.2. Конструкторы типов как функции над типами

АТД-программирование требует несколько иного взгляда на понятие типа, чем принятый в традиционных языках. В последних под типом переменной понимается не более чем некоторая доступная компилятору информация, определяющая, какие машинные команды следует использовать для действий с этой переменной и какой объем памяти следует выделить для ее размещения. В рамках же АТД-программирования мы будем говорить о типе как о некотором объекте периода компиляции. Эти объекты порождаются и уничтожаются в процессе компиляции, они могут иметь имена, над ними допускаются некоторые операции. Как мы увидим далее, объекты-типы могут быть параметрами пакетов (аналогов процедур периода компиляции). Такой подход требует и несколько иного взгляда на сам процесс компиляции. Опытный программист привык представлять себе выполнение программы как протекающий

во времени процесс. На компиляцию же программы многие смотрят как на некоторое «моментальное» событие, состоящее в том, что из текста программы порождается объектный модуль и листинг. В некоторых случаях, например при работе на ПЛ/1, компиляция распадается для программиста на две последовательные фазы: фазу работы препроцессора и собственно фазу компиляции. Мы будем далее рассматривать компиляцию как протекающий во времени процесс, во время которого программа анализируется строка за строкой. Для такого взгляда важно, что язык Паскаль разрабатывался так, чтобы допускалась компиляция за один просмотр исходного текста. Действия, выполняемые компилятором при анализе той или иной конструкции исходной программы, мы будем называть *выполнением* этой конструкции. Это *выполнение*, конечно же, отличается от собственно выполнения исполняемых операторов во время прогона программы. Во время *выполнения* разделов объявлений производятся только следующие основные действия: синтаксический анализ программы и порождение объектов периода компиляции. Имеются два основных класса этих объектов: объекты-типы и объекты-имена. *Объект-тип* создается при компиляции конструктора типа. *Объект-имя* создается, когда в тексте программы встречается новое имя переменной, типа, процедуры или функции. Отношение *переменная принадлежит типу* выражается тем, что в объекте-имени, описывающем данную переменную, хранится ссылка на объект-тип, которому принадлежит переменная. Таким образом, даже если тип не имеет собственного имени, доступ к нему может быть получен, если есть ссылка на него хотя бы в одном объекте-имени, именуемом переменную, функцию и т. п. Объекты периода компиляции уничтожаются при достижении конца того блока программы, где они были порождены. Это объясняет, почему вне процедуры или функции (процедуры и функции являются единственными видами блоков в языке Паскаль) нельзя использовать имена объявленных в ней объектов. Почему нельзя использовать значения объявленных в процедуре переменных после выхода из нее, понятно. Потому, что память под них отводится при входе в процедуру и освобождается при выходе из нее. А вот почему нельзя использовать ссылку на объект-тип, порожденный при компиляции процедуры? Потому, что этот объект-тип уничтожается в конце компиляции процедуры. Хотя это скорее не фундаментальное свойство языка, а ограничение реализации, вводимое из соображений экономии памяти, подобно тому, как динамическое распределение памяти под локальные переменные процедур позволяет повторно использовать одну и ту же область памяти под переменные разных процедур. Хотя для последнего есть и другое обоснование. Динамическое выделение памяти обеспечивает возможность рекурсивного вызова процедур, Языки со статическим распределением

памяти, например Фортран, этой возможности не обеспечивают. Заметим, что предопределенные стандартные типы языка порождаются в начале и существуют в течение всего процесса компиляции.

Рассмотрим теперь, какие же операции допустимы над объектами-типами. Наиболее важная и имеющаяся почти во всех языках операция — это использование типа при объявлении имени. При выполнении ее образуется еще одна ссылка на объект-тип, т. е. возникает отношение *принадлежит типу* между объектом-именем и объектом-типом. Другая группа операций — это порождение новых объектов-типов с помощью конструкторов типов. Удобно представлять себе конструкторы типов как функции периода компиляции, аргументами которых являются ссылки на объекты-типы, а значениями — ссылки на новые объекты-типы. Например, конструктор регулярного типа можно условно записать в виде функции `array(I, E)`, где аргумент `I` есть ссылка на объект-тип, описывающий тип индекса массива, а `E` — ссылка на объект-тип, описывающий тип элементов массива. Значением же функции является ссылка на новый объект-тип, порожденный функцией. Аналогичную функциональную запись могут иметь конструкторы `SET OF` и `FILE OF`: соответственно `set(T)` и `file(T)`. Конструктор комбинированного типа в функциональном выражении имеет переменное число параметров, соответствующее количеству полей записи. Конструктор ограниченного типа будет записываться как `range(T, N1, N2)`, где `N1` и `N2` — константы базового типа `T`. В функциональной записи конструктор

`FILE OF ARRAY[1..10,CHAR] OF SET OF I`

будет выглядеть так:

`file(array(range(INTEGER,1,10),array(CHAR,set(I))))`

Конечно, этот функциональный вид записи конструкторов типа менее нагляден, чем принятый в языке Паскаль. Он вводится только для того, чтобы облегчить определение действия деструкторов типа, рассматриваемых ниже.

Таким образом, конструкторы типа языка Паскаль мы будем трактовать как функции периода компиляции, создающие новый объект-тип и возвращающие ссылку на него. Аргументами и значениями этих функций являются типы. Не имеет аргументов только конструктор скалярного типа. Вернее, он имеет произвольное количество аргументов, но ими являются не типы, а некоторые абстрактные имена, не обозначающие ничего, кроме самих себя,

Итак, имеются четыре группы операций над типами:

- 1) создание (выполняется конструкторами типов);
- 2) присваивание (рассмотрено ниже);

3) уничтожение (автоматическое при выходе из блока);

4) декомпозиция (выполняется деструкторами типов, рассмотренными ниже).

Эти операции выполняются при компиляции заголовков процедур и разделов объявлений. Заметим, что в такой трактовке процесса компиляции составной оператор, входящий в блок (собственно тело блока), при компиляции которого порождается объектный код, является *пустым оператором*. При его обработке никаких действий над объектами периода компиляции не производится.

Рассмотрим более подробно, что является аналогом операции присваивания при выполнении операций над типами. Типу дается собственное имя при компиляции объявления вида

TYPE T=ARRAY[1..10] OF BOOLEAN;

Здесь порождается объект-имя T, которому присваивается ссылка на объект-тип, описывающий регулярный тип, который в свою очередь ссылается на два объекта-типа: ограниченный тип индекса 1..10 и предопределенный тип INTEGER. В дальнейшем «значением» имени T становится эта ссылка на созданный объект. Употребление имени T где бы то ни было в программе поставляет это значение. Заметим, что именно так в этом примере использовано имя INTEGER. Предполагается, что значением этого имени является ссылка на предопределенный объект-тип. В нашей условной функциональной записи это объявление типа T могло бы быть записано так:

T := array(range(INTEGER,1,10),BOOLEAN);

Объявление вида

TYPE Q=T;

выполняет создание объекта-имени, присваивает ему в качестве значения ссылку на тот объект, на который ссылается T. То есть это объявление можно выразить так:

Q := T

После такого *присваивания* эти два имени будут обозначать один и тот же объект-тип. Аналогично *выполняются* и объявления переменных, с той разницей, что если объект-имя описывает переменную, а не тип, то с ним связывается еще некоторая добавочная информация, например относительный адрес переменной. Семантика языков программирования такова, что использование имени обычно обозначает не «значение» имени, а «значение» переменной, обозначаемой этим именем. Однако с точки зрения операций над типами имя типа и имя переменной, принадлежащей этому типу, совершенно равноправны. И то, и другое имя обладает «значением» — ссылкой на объект-тип. Заметим, что, в отличие от переменных

периода выполнения, значения именам присваиваются однократно и в дальнейшем уже не могут быть изменены.

На рис. 1 изображены объекты-имена (изображаемые шестиугольниками) и объекты-типы (изображаемые прямоугольниками), возникающие при компиляции следующей совокупности объявлений:

```
TYPE E = 1..100;
T = ARRAY[1..10] OF E;
VAR A, B: T;
    C: E;
```

Стрелками изображаются связи между ними, т. е. ссылки. Разумеется, мы не раскрываем здесь все содержание объектов, Кроме

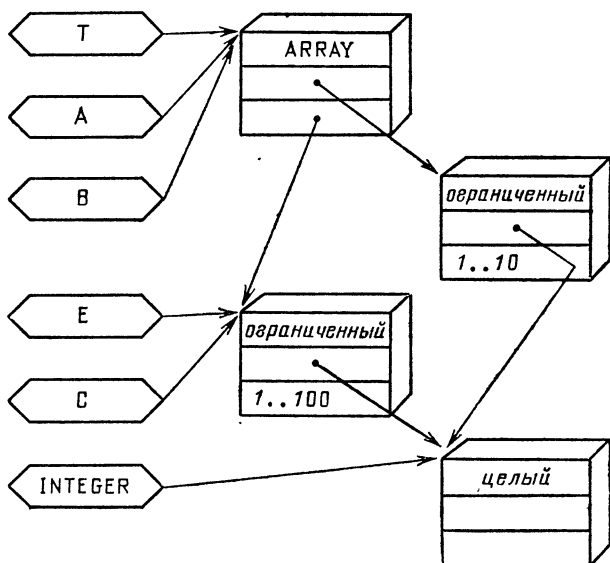


Рис. 1. Связи объектов-имен и объектов-типов

указанной информации и ссылок они содержат и другую информацию, необходимую для обеспечения процесса компиляции. Например, объект-тип содержит требуемый для представления данного типа объем памяти.

### 2.3. Деструкторы типов и функции-константы

Как видно, объекты-типы являются теми «кубиками», из которых строятся сложные типы. Каждый из них однозначно идентифицируется указывающей на него ссылкой. Однако в стандартном Паскале нет средств, позволяющих выделить один из «кубиков», входящих в состав сложного типа, Построение сложных типов

в языке Паскаль всегда идет по принципу «снизу — вверх», т. е. от более простых типов к более сложным. Разумеется, в том случае, когда некоторый простой тип имеет собственное имя, как, например, тип `E` на рис. 1, он может входить одновременно в состав нескольких сложных типов. Однако, как видно из того же рисунка, объект-тип не всегда имеет имя, т. е. не всегда существует ссылающийся на него объект-имя. Как мы увидим впоследствии, даже если это имя и существует, часто использовать его неудобно или нецелесообразно. Поэтому в языке АТ-Паскаль введен новый класс языковых средств — *деструкторы типов*. Как и конструкторы типов, они являются функциями над типами периода компиляции. Их аргументами и значениями являются ссылки на типы. В отличие от конструкторов, деструкторы типов не создают новые объекты-типы. Они лишь позволяют получить ссылку на любой из составляющих сложный тип «кубиков», чтобы использовать ее либо для объявления переменной, либо как аргумент конструктора, порождающего новый тип. Деструкторы типов являются как бы *обратными функциями* по отношению к конструкторам. Они логично дополняют возможности последних, обеспечивая программисту полную свободу манипулирования типами.

В отличие от конструкторов, деструкторы являются функциями не только по сути, но и синтаксически. Деструкторы выглядят как обращения к функциям, аргументы которых могут иметь некоторый специальный вид. При построении аргументов деструкторов могут использоваться так называемые *селекторы аргументов*.

Определены четыре деструктора типов. Возвращаемые деструкторами значения будем определять с использованием функциональной записи конструкторов. Поскольку деструкторы являются элементами языка, их имена будем записывать прописными буквами, в отличие от имен конструкторов, для задания которых в функциональной записи используются строчные буквы,

#### 1. Деструктор `ELEM(T)`.

Справедливы следующие тождества:

$$\begin{aligned} \text{ELEM}(\text{array}(K, Q)) &= Q; \\ \text{ELEM}(\text{packed}(\text{array}(K, Q))) &= Q; \\ \text{ELEM}(\text{set}(Q)) &= Q; \\ \text{ELEM}(\text{file}(Q)) &= Q. \end{aligned}$$

Таким образом, деструктор `ELEM` возвращает ссылку на тип элемента массива, множества или файла. Он является *обратной функцией* по отношению к конструкторам `ARRAY`, `SET` и `FILE`,

#### 2. Деструктор `INDEX(T)`.

$$\begin{aligned} \text{INDEX}(\text{array}(K, Q)) &= K; \\ \text{INDEX}(\text{packed}(\text{array}(K, Q))) &= K. \end{aligned}$$

Деструктор возвращает ссылку на тип индекса массива.



### 3. Деструктор BASE(T).

BASE(ref(T)) = T;

BASE(packed(T)) = T;

BASE(range(T, N1, N2)) = T.

Деструктор возвращает ссылку на базовый тип некоторого типа T.

### 4. Деструктор TYP(X).

Если аргумент есть имя константы, переменной или функции, то значение деструктора TYP — ссылка на тип этой константы, переменной или функции. Если аргумент есть ссылка на тип, то значение деструктора тождественно равно аргументу.

Рассмотрим простые примеры применения деструкторов типа, Если R есть массив, объявленный так:

```
VAR R:ARRAY[1..10] OF CHAR;
```

то значения деструкторов будут следующие:

ELEM(R) — тип элементов массива (ссылка на тип CHAR);

INDEX(R) — тип индекса массива (ссылка на тип 1..10).

Аргументом деструктора является ссылка на тип, получающаяся при вычислении специального выражения. Аргумент может быть либо простым аргументом, либо аргументом с селекторами. *Простой аргумент* — это либо деструктор типа, либо имя константы, переменной или функции. В этом случае значением аргумента является ссылка на тип константы, переменной или функции. *Аргумент с селекторами* — это простой аргумент, за которым идет один или несколько так называемых селекторов. Имеются четыре вида селекторов.

1. *Селектор массива* служит для перехода от регулярного типа к типу его компонент. Если T — ссылка на тип массива, то T[ ] — ссылка на тип элементов массива. Для доступа к типам элементов многомерного массива вместо записи вида T[ ] [ ] [ ] можно использовать такую: T[,]. То есть T[ ] — тип элементов одномерного массива, T[, ] — двумерного, T[, ] — трехмерного и т. д. Таким образом, для регулярных типов действие селектора совпадает с действием деструктора ELEM:

ELEM(ELEM(T)) эквивалентно TYP(T[,]).

2. *Селектор поля записи* служит для перехода от комбинированного типа к типу поля. Он состоит из символа точка и имени поля. Например, если P — тип записи, содержащей поле C, то значением деструктора TYP(R.S) является тип поля S.

3. *Селектор указателя* (символ @) служит для перехода от типа указателя к его базовому типу. То есть для типа указателя

его действие совпадает с действием деструктора BASE:

BASE(T) эквивалентно TYP(T@).

4. *Селектор параметра* имеет вид (,,) и может идти только след за именем процедуры или функции. Он возвращает ссылку на тип параметра процедуры или функции. Номер параметра на единицу больше количества запятых в селекторе, т. е. ( ) — первый параметр, (,) — второй параметр, (,,) — третий параметр и т. д.

При вычислении некоторого деструктора сначала вычисляется его аргумент, а затем сам деструктор. Вычисление аргумента производится слева направо, причем каждый селектор действует как одноместная постфиксная операция. Деструкторы типов могут использоваться везде, где синтаксис языка Паскаль допускает использование конструкторов типов, т. е. в объявлениях типов, констант, переменных, функций, параметров процедур и функций. Заметим, что аргумент деструктора часто совпадает с обозначением некоторой переменной, в котором опущены индексные выражения. Например, аргумент S[ ]@.B обозначает тип переменной, обозначаемой в программе так: S[J]@.B. Однако такое соответствие существует не всегда.

Рассмотрим пример вычисления сложного деструктора, Предположим, что есть совокупность объявлений

```
TYPE R=RECORD
      A:INTEGER;
      B:SET OF CHAR
    END;
    T=ARRAY[1..10] OF @R;
VAR C:T;
```

Вычисляемый деструктор — ELEM(C[ ]@.B). Его вычисление производится в следующей последовательности.

Вычисляемое подвыражение	Значение
1. C	тип T
2. C[ ]	@R
3. C[ ]@	R
4. C[ ]@.B	SET OF CHAR
5. ELEM(C[ ]@.B)	CHAR

Рассмотрим другие примеры деструкторов типов.

Пример 1. Предположим, в программе есть объявление типа

```
TYPE W=ARRAY[1..10] OF
      RECORD
        S:SET OF 'A'..'F';
        A:ARRAY[BOOLEAN] OF INTEGER;
```

```

R:RECORD
  J:0..100;
  X:REAL
END

```

END;

тогда значения деструкторов будут такими:

INDEX(W)	— 1..10
ELEM(W[ J.S)	— 'A'..'F'
BASE(ELEM(ELEM(W).S))	— CHAR
INDEX(W[ J.A)	— BOOLEAN
ELEM(W[ J.A)	— INTEGER
TYP(W[ J.A[ J)	— INTEGER
TYP(W[ J.R.J)	— 0..100
BASE((W[ J.R.J)	— INTEGER
TYP(W[ J.R.X)	— REAL

Пр и м е р 2. Предположим, что в программе есть следующее объявление процедуры Р с заголовком:

```

PROCEDURE P(S:SET OF 1..20;
  FUNCTION Q(Y:REAL):INTEGER;
  VAR A:ARRAY[1..5,1..10]
    OF BOOLEAN);

```

тогда значения деструкторов будут такими:

TYP(P( ))	— SET OF 1..20
ELEM(P( ))	— 1..20
BASE(ELEM(P( )))	— INTEGER
INDEX(P(,))	— 1..5
INDEX(P(,)[ J)	— 1..10
ELEM(P(,)[ J)	— BOOLEAN
TYP(P(,))	— INTEGER
TYP(P(,)( ))	— REAL

Так называемые *функции-константы* языка занимают промежуточное положение между обычными функциями и деструкторами типов (функциями над типами). С одной стороны, их аргументами являются ссылки на типы и строятся они по правилам аргументов деструкторов. С другой стороны, значением функции-константы является константа и использоваться она может везде, где синтаксис языка допускает использование вызовов обычных функций, т. е. в выражениях. Как и деструкторы, функции-константы являются функциями периода компиляции, так как их вычисление производится во время компиляции и в процессе выполнения программы это значение уже не может измениться. Определены три функции-константы.

1. FIRST(S); если S есть ссылка на скалярный или ограниченный тип, то значение функции-константы равно минимальному значению, которое могут принимать переменные данного типа. Тип значения функции-константы есть тип S.

2. LAST(S) аналогична FIRST, но возвращает максимальное значение.

3. SIZE(S), S — любой тип, функция возвращает размер памяти в байтах, отводимой под переменные этого типа.

Пример 3. Если объявление типа W такое же, как в примере 1, то значения функций-констант будут следующими:

FIRST(INDEX(W))	1
LAST(INDEX(W))	10
FIRST(INDEX(W[ ] .A))	FALSE
LAST(INDEX(W[ ] .A))	TRUE
FIRST(ELEM(W[ ] .S))	'A'
LAST(ELEM(W[ ] .S))	'F'
SIZE(W[ ] .A[ ])	4
SIZE(W[ ] .R .X)	8

Заметим, что тип функций-констант совпадает с типом аргумента. Наиболее частое применение этих функций — организация цикла по множеству значений индекса массива. Например, если V есть некоторый регулярный тип, то допустимы следующие конструкции:

```
VAR J:INDEX(V);
```

```
  ..  
FOR J := FIRST(J) TO LAST(J) DO...
```

или

```
FOR J := FIRST(INDEX(V)) TO LAST(INDEX(V)) DO...
```

Эти операторы будут действовать правильно, обеспечивая перебор всех допустимых значений индекса массива, при любом типе индексов.

## 2.4. Ссылочная идентичность типов

В любом языке программирования со строгим контролем типов и возможностью определения новых типов данных особую важность приобретает вопрос об эквивалентности типов. Как правило, требуется, чтобы левая и правая части оператора присваивания имели эквивалентные типы и чтобы совпадали типы формальных и фактических параметров процедур. В различных реализациях языка Паскаль исторически сложились два подхода к определению эквивалентности или, как принято называть в языке Паскаль, идентич-

ности типов: *структурная* и *именная*. При структурном подходе, который применялся в ранних компиляторах языка Паскаль, два типа считаются идентичными, если переменные, обладающие этими типами, имеют одинаковую структуру. Например, в случае регулярного типа должны быть идентичными типы индексов и совместимыми типы элементов. В случае комбинированного типа должны совпадать количества полей и быть попарно идентичными их типы. Понятие совместимости типов менее строго, чем понятие идентичности. Оно определяет типы, которые могут участвовать в операции присваивания даже тогда, когда они неидентичны. Например, если две переменные объявлены так:

```
VAR N:0..100;  
    M:1..10;
```

то присваивание  $N:=M$  допустимо всегда, а  $M:=N$  — только тогда, когда значение  $N$  лежит в диапазоне 1..10. Эти две переменные обладают хотя и неидентичными, но совместимыми типами.

При именной идентичности две переменные считаются обладающими идентичными типами, если они либо объявлены с использованием одного и того же имени типа, либо объявлены с разными именами типов, причем одно из них определено через другое с помощью объявления вида

```
TYPE T2=T1;
```

Этот подход, хотя и является более строгим, накладывает, на наш взгляд, чересчур сильные ограничения, заставляя программиста давать уникальные имена почти каждому используемому типу. Это загромождает программу, делая ее менее понятной. Например, если есть объявление

```
VAR A:ARRAY[1..N,1..M] OF REAL;  
    B:ARRAY[1..M] OF REAL;
```

то допустимое при структурной идентичности присваивание  $B:=A[J]$  некорректно в смысле именной идентичности. Чтобы сперировать с переменной, идентичной по типу строке матрицы, надо объявить тип строки отдельно. Например, так:

```
TYPE ROW=ARRAY[1..M] OF REAL;  
VAR A:ARRAY[1..N] OF ROW;  
    B:ROW;
```

При этом программисту, читающему программу, с первого взгляда не видно, что  $A$  — двумерный массив. Ему надо сначала отыскать в тексте программы (возможно, совсем в другом месте) объявление типа  $ROW$ . Анализ этих двух подходов к понятию идентичности дан в послесловии к [8].

Основываясь на введенной выше трактовке типов как особых объектов периода компиляции, можно дать иное определение идентичности, которое и используется в языке АТ-Паскаль. Будем называть эту идентичность *ссылочной*.

**О п р е д е л е н и е   с с ы л о ч н о й   и д е н т и ч н о с т и .**  
Два объекта программы обладают идентичными типами, если описывающие их объекты-имена ссылаются на один и тот же объект-тип.

Поскольку деструкторы типа вырабатывают именно ссылки на составные части сложных типов, переменная, объявленная с помощью какого-либо деструктора, имеет тип, идентичный соответствующей части сложного типа. Например,

```
VAR A:ARRAY[1..N,1..M] OF REAL;  
    B:ELEM(A);
```

Здесь переменная В имеет тип, идентичный (в смысле ссылочной идентичности) типу строки матрицы А и, следовательно, присваивание В:=А[J] допустимо.

Если по каким-либо причинам требуется дать имя составной части сложного типа, выделяемой некоторым деструктором, это легко можно сделать. Например,

```
TYPE ROW = ELEM(A);  
      IND1 = INDEX(A);  
      IND2 = INDEX(ELEM(A));
```

Заметим, что именная идентичность является частным случаем ссылочной. Действительно, если несколько объектов объявляются с использованием одного и того же имени типа (т. е. являются, по определению, идентичными в смысле именной идентичности), то одна и та же ссылка на этот объект-тип присваивается всем описывающим их объектам-именам. Следовательно, они будут идентичными в смысле ссылочной идентичности. При выполнении конструкции

```
TYPE T2 = T1;
```

ссылка на объект-тип, на который ссылается имя Т1, присваивается объекту-имени Т2. Следовательно, если две переменные объявлены так:

```
VAR A:T1;  
    B:T2;
```

(их типы, по определению, идентичны в смысле именной идентичности), объектам-именам, описывающим переменные, присваивается ссылка на один и тот же объект-тип. Следовательно, они будут идентичными в смысле ссылочной идентичности. Таким образом,

показано, что именная идентичность типов двух объектов влечет за собой их ссылочную идентичность. Обратное утверждение неверно. Именная идентичность типов является частным случаем ссылочной, поэтому изменение определения идентичности типов по сравнению со стандартом языка Паскаль не приведет к потере корректности программ, написанных с использованием именной идентичности.

## 2.5. Примеры применения деструкторов типов

Рассмотрим сначала простейшие примеры применения деструкторов типа. Присвоение имени каждому новому типу удобно далеко не всегда. Например, видя в программе объявление вида

```
VAR A:T;
```

программист ничего не может сказать о типе переменной, пока не найдет где-то в другом месте текста объявление

```
TYPE T=ARRAY[1..4] OF INTEGER;
```

Удобнее было бы объявлять переменную так:

```
VAR A:ARRAY[1..4] OF INTEGER;
```

Однако, если переменная объявлена с использованием своего собственного конструктора типа, становится невозможным объявить какую-либо другую переменную с типом, идентичным типу переменной A. Например, если процедура имеет заголовок

```
PROCEDURE P(VAR B:ARRAY[1..4] OF INTEGER);
```

ее вызов вида P(A) с использованием переменной A как параметра некорректен, так как типы переменной A и параметра B неидентичны. Без использования деструкторов типа языка АТ-Паскаль невозможно объявить переменную с типом идентичным типу параметра процедуры P. Поэтому в стандартной версии языка Паскаль в объявлениях параметров процедур и функций используются только имена типов, но не произвольные конструкторы. В стандартной версии языка приведенный выше заголовок процедуры некорректен. В языке АТ-Паскаль он допустим. Заметим, что типы переменной A и параметра B неидентичны как в смысле именной, так и ссылочной идентичности. Однако, используя деструктор типа в объявлениях

```
VAR C:TYP(A);
```

```
PROCEDURE P(VAR B:TYP(A));
```

мы обеспечиваем правильность операторов C:=A и P(A). Здесь имя A обозначает одновременно и тип, и конкретный экземпляр

переменной данного типа. Такой прием может сделать текст программы более понятным за счет сокращения количества имен объектов.

В языке Паскаль переменные, являющиеся параметрами циклов, должны быть объявлены в той же процедуре, где используется цикл. Например,

PROCEDURE EXAMPLE;

VAR I:1..N;

J:1..M;

...

FOR I := 1 TO N DO

FORJ := 1 TO M DO A[I,J] := 0;

...

Однако, видя перед собой только текст этой процедуры, нельзя с уверенностью сказать, будет ли правильно обнулен весь массив и не выйдут ли значения индексов за его границы. Для этого надо сначала найти объявление массива A, затем объявление его типа. Затем надо убедиться, что константы M и N, использованные в объявлении типа массива, те же самые, что и константы, использованные в процедуре. Например, они могли бы быть переопределены на промежуточных блочных уровнях. Как видим, такой простой фрагмент таит в себе довольно много потенциальных источников ошибок. С использованием деструкторов этот фрагмент может быть переписан так:

PROCEDURE EXAMPLE;

VAR I:INDEX(A);

J:INDEX(A[ I]);

...

FOR I,J DO A[I,J] := 0;

...

Здесь можно быть уверенным в том, что независимо от того, как объявлен массив A, он будет обнулен полностью и корректно. Заметим, что не надо даже знать типы индексов массива. В языке АТ-Паскаль, если в операторе цикла с параметрами не указаны пределы изменения параметра, предполагается, что он пробегает все допустимые его типом значения, То есть конструкция

FOR I,J DO...

эквивалентна следующей:

FOR I := FIRST(I) TO LAST(I) DO

FOR J := FIRST(J) TO LAST(J) DO...

Рассмотрим теперь применение деструкторов типа при конструировании так называемых полиморфных процедур. *Полиморфными*



*процедурами* называются такие процедуры, которые в максимальной степени независимы от конкретных типов обрабатываемых ими данных. Полиморфная процедура настраивается на те или иные типы данных при компиляции. Полиморфные процедуры называют иногда родовыми (*generic*) процедурами, однако мы этот термин применять не будем. Иногда, например в языке Ада, процесс настройки полиморфной процедуры на нужные типы данных называется конкретизацией. Заметим, что объявления так называемых семейств точек входов в процедуры языка ПЛ/1 не являются аппаратом, обеспечивающим программирование полиморфных процедур. Эти средства позволяют в зависимости от типов фактических параметров в точке вызова произвести обращение к той или иной процедуре семейства. Но самих процедур должно быть столько, сколько может быть различных сочетаний типов фактических параметров.

Возьмем для начала простой пример: функцию суммирования массива. Разумеется, любой, даже начинающий программист, знакомый с каким-либо языком программирования, например с ПЛ/1 или Фортраном, без труда напишет такую функцию за несколько минут. Но заметим, что первый вопрос, который он, видимо, задаст, будет: «Каков тип элементов массива — целый, вещественный, двойной точности?» Вопросы о типе индексов не возникает, так как в этих языках индексы могут быть только целочисленными. Программист, работающий на языке Паскаль, спросит еще и о типе индекса, причем просто имени типа ему окажется мало. Он должен будет получить еще исчерпывающую информацию о том, что это за тип, чтобы организовать перебор всех значений индекса. Если же потребовать написать функцию суммирования массива, зная только имя типа и то, что это регулярный тип, над компонентами которого допустима операция сложения, программист, работающий на традиционных языках, видимо, скажет, что это невозможно. Ведь надо же указать тип функции, идентичный типу элементов массива. Надо объявить индексную переменную с таким же типом, как тип индекса суммируемого массива. Используя аппарат деструкторов типа языка АТ-Паскаль, такую процедуру написать не только можно, но и довольно легко. Вот она:

```
FUNCTION VECTSUM(V:VECTOR):ELEM(VECTOR);
  VAR S:ELEM(VECTOR);
      J:INDEX(VECTOR);
  BEGIN
    S := 0;
    FOR J DO S := S+V[J];
    VECTSUM := S
  END;
```

Тип VECTOR в программе, использующей эту функцию, может, например, быть объявлен так:

```
TYPE VECTOR=ARRAY[0..N] OF REAL;
```

или так:

```
TYPE VECTOR=ARRAY[(A,B,C,D,E,F)] OF INTEGER;
```

или так:

```
TYPE VECTOR=ARRAY['A'..'Z'] OF -100..100;
```

Заметим, что в языке Паскаль операция «+» определена и над множествами. В этом случае она обозначает объединение множеств. Можно немного изменить данную функцию, чтобы она одновременно могла быть использована для объединения всех элементов массива множеств. Для этого надо устранить из процедуры присваивание  $S:=0$ , которое явно диктует, что переменная S должна иметь тип INTEGER или REAL. Это можно сделать, например, так:

```
S := V[FIRST(J)];
```

```
FOR J := SUCC(FIRST(J)) TO LAST(J) DO  
  S := S+V[J];
```

Функция SUCC языка Паскаль возвращает следующее значение скалярного или ограниченного типа. Эта функция является полиморфной, она применима к любому скалярному или ограниченному типу, и тип ее совпадает с типом аргумента. После такой модификации процедуры тип VECTOR может быть, например, таким:

```
TYPE VECTOR=ARRAY[1..100] OF SET OF 1..10;
```

Как видим, функция VECTSUM действительно является полиморфной. Она применима к потенциально бесконечному множеству типов: к массивам с произвольными типами индексов и целым, вещественным или множественным типом элементов.

Второй пример полиморфной функции также весьма прост: поиск максимального элемента в массиве. Заметим, что, хотя мы рассматриваем только примеры функций, процедуры могут быть полиморфными с тем же успехом. Для массивов, над элементами которых допустима операция сравнения, написать такую полиморфную функцию не составляет труда. Усложним немного задачу, потребовав, чтобы она действовала над массивами любых типов. Поскольку операции сравнения применимы не ко всем типам, для преобразования элементов массива в сравнимые значения будет использоваться специальная функция KEY. Ее тип может быть любым, допускающим сравнения, т. е. целым, вещественным, логическим, символьным, скалярным или ограниченным. Это может

также быть строка символов. Предполагается, что эта функция и тип VECTOR объявлены в программе раньше того места, где включен текст самой функции следующего вида:

```
FUNCTION MAXINDEX(A:VECTOR):INDEX(VECTOR);  
  VAR J,MAX :INDEX(VECTOR);  
      V,MAXV:TYP(KEY);  
  BEGIN  
    MAX := FIRST(J);  
    MAXV := KEY(A[MAX]);  
    FOR J := SUCC(FIRST(J)) TO LAST(J) DO  
      BEGIN  
        V := KEY(A[J]);  
        IF V > MAXV THEN  
          BEGIN MAXV := V; MAX := J END  
        END;  
    MAXINDEX := MAX  
  END;
```

Обратим внимание на использование деструктора TYP для объявления переменных V и MAXV, которые должны иметь тип, идентичный типу функции, преобразующей элементы массива в сравнимые значения. Какой это будет тип, при написании процедуры мы не знаем.

Рассмотрим пример использования функции MAXINDEX. Предположим, что тип VECTOR объявлен так:

```
TYPE VECTOR=ARRAY[M..N] OF  
  RECORD  
    A:REAL;  
    B,C:INTEGER;  
    D:SET OF 0..20  
  END;
```

Мы хотим использовать функцию MAXINDEX для нахождения в массиве Q типа VECTOR элемента с максимальной суммой полей A и C. Тогда функция KEY должна иметь следующий вид:

```
FUNCTION KEY(X:ELEM(VECTOR)):REAL;  
  BEGIN  
    KEY := X.A + X.C  
  END;
```

Вызов функции производится обычным образом:

```
K := MAXINDEX(Q);
```

Хотя сами по себе полиморфные процедуры с использованием аппарата деструкторов типа записываются весьма изящно, при попытке их практического использования мы сталкиваемся с рядом вопросов. А что если тип, для которого мы хотим применить функцию VECTSUM, называется не VECTOR, а иначе? Что делать, если мы хотим в одном и том же блоке применить функцию VECTSUM к массивам разных типов? Что делать, если в нашей программе уже есть объект с именем VECTSUM и менять это имя нежелательно? Ответы на эти вопросы дает аппарат параметризуемых пакетов языка АТ-Паскаль, описываемый в следующей главе.

## ГЛАВА 3

## ПАКЕТЫ

### 3.1. Понятие пакета

Под термином *пакет*, заимствованным из терминологии языка Ада, мы будем понимать совокупность объектов программы, объединение которых в один пакет диктуется логикой программы и преследует определенную цель. Под объектом программы мы понимаем здесь константы, переменные, типы, процедуры и функции. Например, имеет смысл объединение в пакет набора процедур комплексной арифметики или набора процедур для работы с матрицами, однако объединение в один пакет процедуры умножения комплексных чисел и процедуры сложения вещественных матриц, видимо, бессмысленно, так как они действуют над объектами существенно разных типов и не имеют друг с другом ничего общего. В состав пакета могут входить не только процедуры и функции, но и константы, типы, переменные, а также другие пакеты. Чтобы подчеркнуть различие между пакетом языка АТ-Паскаль и таким известным понятием, как пакет прикладных программ, заметим, что пакет языка АТ-Паскаль может вовсе не содержать в себе процедур. В качестве пакета может быть оформлена группа объявлений переменных или даже констант. Однако наиболее употребительный состав пакета, применяемый в АТД-программировании, таков: объявление типа и набор процедур и функций для выполнения некоторого множества действий над объектами этого типа. Именно такие пакеты используются для определения абстрактных типов данных.

Синтаксис пакета языка АТ-Паскаль тривиально прост. Пакет представляет собой просто совокупность объявлений на языке АТ-Паскаль. В некоторых случаях пакет может быть снабжен *заголовком*, синтаксис которого близок к синтаксису заголовка процедуры (см. приложение). Заголовок необходим только тогда, когда пакет имеет так называемые *позиционные параметры*. Пакеты хранятся во внешней памяти ЭВМ, где каждый пакет имеет уникаль-

ное имя, называемое далее *внешним именем пакета*. Текст пакета включается в текст использующей его программы во время компиляции. Аппарата раздельной компиляции пакетов язык АТ-Паскаль не предусматривает. Раздельная компиляция допускается только для процедур. Этот недостаток ограничивает применение языка АТ-Паскаль программами средней сложности объемом не более 10—20 тысяч строк.

Несколько слов о терминологии. Конечно, использование термина «пакет» не совсем удачно хотя бы потому, что даже в вычислительной технике этот термин имеет по крайней мере четыре значения, не говоря об общеупотребительных. Это — пакет прикладных программ, пакет магнитных дисков, пакет заданий и пакет как единица межмашинного обмена в сетях ЭВМ. Такая путаница в терминологии происходит из-за заимствований терминов из английского языка (заметим, что в нем эти термины различны: соответственно *package, pack, batch, packet*). В ряде языков программирования понятие, соответствующее нашему понятию пакета, обозначается иначе, например, кластер — в языке CLU, форма — в языке Alphard. Однако в языке Ада, который, видимо, получит распространение в ближайшем будущем в нашей стране, принят термин *package*, переведенный как «пакет». Чтобы подчеркнуть близость аппарата языков Ада и АТ-Паскаль и чтобы не вводить новых терминов для обозначения того, что уже имеет имя, пусть и неудачное, в языке АТ-Паскаль используется термин «пакет».

### 3.2. Присоединение и вложение пакетов

Пакет, хранящийся в библиотеке, может быть включен в программу двумя способами: *присоединением* и *вложением*. Для включения пакета в программу его внешнее имя указывается в так называемом *разделе присоединения*, который может быть помещен среди других разделов объявлений любого блока или пакета, причем таких разделов может быть несколько (синтаксис см. в приложении). В дальнейшем для краткости под термином *блок* мы будем понимать как собственно блок языка Паскаль, так и пакет. Блок в обычном понимании будем называть *процедурным блоком*, так как в языке Паскаль нет других видов блоков, кроме процедур и функций.

В простейшем случае раздел присоединения имеет вид

USE A;

где А — внешнее имя пакета. Этот вид раздела присоединения обеспечивает включение пакета в режиме присоединения. Блок, имеющий раздел присоединения такого вида, компилируется в точности так, как если бы текст пакета (без заголовка) был помещен

в текст программы вместо раздела присоединения. Такой раздел присоединения эквивалентен хорошо знакомому большинству программистов оператору INCLUDE (в языках Ассемблера, ПЛ/1, Кобол, Паскаль и др.). Разница только в том, что оператор INCLUDE позволяет поместить в любое место текста произвольный фрагмент программы, а раздел присоединения позволяет включать только синтаксически законченные части (объявления) среди разделов объявлений блока. При таком включении все объявленные в пакете объекты становятся непосредственно доступными в данном блоке. Под непосредственной доступностью мы будем понимать возможность обратиться к объекту, просто указав его имя. Далее мы будем называть множество объявленных в блоке (пакете) имен *контекстом данного блока* (пакета). Используя этот термин, можно сказать, что в случае присоединения пакета его контекст объединяется с контекстом объемлющего блока (присоединяется к нему). Присоединение — это наиболее простой, но ограниченный способ использования пакетов, так как при его применении возможны *конфликты имен*. Под конфликтом имен будем понимать объявление в одном и том же блоке двух разных объектов под одним и тем же именем. Например, если пакет А содержит объявление переменной X, а в блоке, содержащем раздел присоединения этого пакета, уже имеется объявление константы X, то присоединение пакета А невозможно.

Для предупреждения возможных конфликтов имен применяется другой способ использования пакетов — вложение. При вложении пакета его контекст не присоединяется к контексту включающего блока, а остается изолированным от него. Будем называть такой пакет вложенным. Для вложения пакета раздел присоединения имеет вид

USE P=A;

где А — по-прежнему внешнее имя пакета, а Р — так называемое имя контекста пакета. *Имя контекста* — это любое имя, не входящее в конфликт с другими объявленными в данном блоке именами. Имя контекста присоединяется к контексту блока. Оно служит как бы именем данного экземпляра пакета. Один и тот же пакет может быть вложен в блок с разными именами контекстов, при этом конфликтов имен не происходит, несмотря на то, что в каждом экземпляре пакета объявлены одни и те же имена. Имена объектов из вложенного пакета не являются непосредственно доступными во включающем блоке. Однако доступ к ним все же может быть получен через имя контекста, поэтому будем говорить, что они доступны косвенно.

С точки зрения видимости имен вложенный пакет весьма напоминает обычный блок, как он понимается в языках с блочной струк-





косвенно. Как мы увидим в дальнейшем, в пакете могут быть объявлены так называемые локальные объекты, доступ к которым извне пакета невозможен, точно так же как и к объектам процедурного блока.

### 3.3. Доступ к объектам вложенного пакета

В языке АТ-Паскаль определены пять способов доступа к объектам вложенного пакета. Рассмотрим сейчас три из них. Заметим, что первые два практически совпадают со способами доступа к полям записей.

1) *Уточненные имена.* Конструкция, состоящая из имени контекста пакета и имени объекта из этого пакета, разделенных точкой, называется уточненным именем. Уточненное имя обеспечивает доступ к конкретному объекту из некоторого вложенного пакета, имя контекста которого известно. Например, к объектам X и Y на рис. 2 в теле блока можно обращаться так: R.X и R.Y. В качестве имени пакета в свою очередь может выступать уточненное имя, т. е. уточнение может быть многократным. Такая ситуация может случиться, когда вложенный пакет имеет раздел присоединения.

Рассмотрим пример:

```
PACKAGE P1;
```

```
VAR S:REAL;
```

```
...
```

```
PACKAGE P2;
```

```
USE Q=P1;
```

```
...
```

```
USE R=P2;
```

В теле пакета P1 к переменной S можно обращаться непосредственно, в теле пакета P2 — через уточненное имя Q.S, в теле внешнего включающего блока — через уточненное имя R.Q.S.

2) *Оператор присоединения.* Действие оператора присоединения языка Паскаль WITH—DO расширено на вложенные пакеты. Если в операторе указано имя контекста пакета, то все объекты этого контекста становятся непосредственно доступными на период компиляции оператора. Например, к переменным пакета на рис. 2 можно обратиться так:

```
WITH R DO X := Y; — — ВМЕСТО R.X := R.Y
```

При этом доступ к переменной X из включающего блока будет временно закрыт.

3) *Раздел присоединения.* Если в разделе присоединения некоторого блока указывается уточненное имя из ранее вложенного



### 3.4. Ключевые параметры пакетов

Аппарат параметризации пакетов позволяет при включении пакета настроить его на использование тех или иных ранее объявленных имен, аналогично тому, как при вызове процедуры мы настраиваем ее на использование тех или иных ранее вычисленных значений. Этот процесс иногда, например в языке Ада, называют *конкретизацией пакета*. Пакеты могут обладать параметрами двух видов: ключевыми и позиционными. Аппарат ключевых параметров более прост, поэтому сначала мы рассмотрим его. Имена ключевых параметров никак не специфицируются в заголовке пакета, поэтому ключевые параметры могут использоваться даже для параметризации пакетов без заголовков. Ключевыми параметрами пакета могут быть только константы или типы. Для записи фактических ключевых параметров также не вводится никакого специального синтаксиса. *Ключевые параметры пакета* — это просто множество объявлений констант и типов, заданное в скобках вслед за внешним именем пакета в разделе присоединения, причем точка с запятой после последнего объявления не ставится. Семантика обработки ключевых параметров весьма проста. Объявления, сделанные в списке фактических ключевых параметров, обрабатываются до начала обработки разделов объявлений пакета, причем в случае вложения блока объявляемые имена присоединяются к контексту пакета, а не к контексту включающего блока. Если имя константы или типа, объявленное в параметрах пакета, совпадает с именем константы или типа, объявленным в пакете, то первое объявление как бы замещает второе. Таким образом, аппарат ключевых параметров позволяет добавлять к пакету произвольные константы и типы или замещать объявления, сделанные в пакете. Последние в этом случае играют роль «значений по умолчанию».

Например, если начало текста некоторого пакета выглядит так:

```
PACKAGE P;  
CONST N=10; M=20;  
TYPE T=INTEGER;
```

...

то раздел присоединения этого пакета с параметрами может иметь вид

```
USE P(CONST N=15;TYPE T=REAL);
```

При этом объявления имен N и T из пакета будут замещены, а значение константы M останется равным 20.

Уже этот достаточно простой аппарат позволяет ответить на вопросы о практическом использовании полиморфных процедур,

поставленные в конце главы 2. Рассмотрим, как решаются эти проблемы, предполагая, что текст полиморфной процедуры VECTSUM, приведенный в п. 2.5, помещен без каких бы то ни было изменений в библиотеку пакетов под внешним именем VECTSUM.

В о п р о с 1: Что делать, если регулярный тип, для которого мы хотим применить функцию VECTSUM, называется не VECTOR, а иначе?

О т в е т

```
TYPE V=ARRAY[0..M] OF INTEGER;
USE VECTSUM(TYPE VECTOR=V);
VAR X:V; S:INTEGER;

. . .
S := VECTSUM(X);
```

В о п р о с 2: Что делать, если мы хотим в одном и том же блоке применить функцию VECTSUM к массивам разного типа?

О т в е т:

```
TYPE V1=ARRAY[0..M] OF INTEGER;
      V2=ARRAY[-10..10] OF REAL;
USE VS1=VECTSUM(TYPE VECTOR=V1),
      VS2=VECTSUM(TYPE VECTOR=V2);
VAR X1:V1;
      X2:V2;
      S1:ELEM(V1);
      S2:ELEM(V2);

. . .
S1 := VS1.VECTSUM(X1);
WITH VS2 DO S2 := VECTSUM(X2);
```

Поскольку пакеты вложенные, никакого конфликта имен не происходит. Разные способы доступа к имени функции VECTSUM использованы в чисто иллюстративных целях. Можно было бы не объявлять типы V1 и V2 отдельно, а сделать это в разделе присоединения так:

```
USE VS1=VECTSUM
      (TYPE VECTOR=ARRAY[0..M] OF INTEGER),
      VS2=VECTSUM
      (TYPE VECTOR=ARRAY[-10..10] OF REAL);
```

В этом случае вместо имен V1 и V2 везде надо было бы использовать уточненные имена VS1.VECTOR и VS2.VECTOR.

В о п р о с 3: Что делать, если в нашей программе уже есть объект с именем VECTSUM и по каким-либо причинам мы не можем изменить его имя?

О т в е т: Использовать вложенный пакет и обращаться к функции VECTSUM из пакета с помощью уточненного имени, а к другому объекту VECTSUM непосредственно.

### 3.5. Позиционные параметры пакетов

Имена формальных позиционных параметров пакета перечисляются в заголовке пакета, синтаксис которого близок к синтаксису заголовка процедуры (см. приложение). Вместо типов параметров, как это указывается в заголовке процедуры, задаются так называемые виды параметров. Позиционный параметр может иметь один из четырех возможных видов: параметр-константа, параметр-тип, входной параметр-имя и выходной параметр-имя. Вид параметра задается одним из следующих зарезервированных слов: соответственно CONST, TYPE, INOUT и OUT. Список фактических позиционных параметров предшествует ключевым параметрам, если они есть. Задавать значения всех указанных в заголовке позиционных параметров не обязательно. Пропущенное значение позиционного параметра в конце списка фактических параметров просто опускается, в начале или в середине списка, как обычно, отмечается запятыми. Здесь и далее под входными параметрами пакета мы будем понимать объекты, которые используются в пакете, но не определяются в нем, а под выходными — те, что определяются в пакете.

Механизм подстановки позиционных параметров зависит от их вида. Для каждого вида он иной. Рассмотрим эти механизмы.

1. П а р а м е т р - к о н с т а н т а. Соответствующим ему фактическим параметром должна быть константа. Параметр обрабатывается так, как если бы к контексту пакета была присоединена константа с именем, равным имени формального параметра, и значением, равным значению фактического параметра. Если в пакете объявлена константа с таким именем, то это объявление замещается, оно может играть роль «значения по умолчанию», если фактический параметр не задан. Например, если заголовок пакета имеет вид

```
PACKAGE P1 (M,N:CONST);
```

то его присоединение вида

```
USE P1 (100,'*');
```

эквивалентно

```
USE P1 (CONST M=100;N='*');
```

2. П а р а м е т р - т и п обрабатывается так же, как и параметр-константа, но допустимыми значениями фактического параметра являются типы. Например,

```
PACKAGE P2 (Q,S:TYPE);
TYPE S = INTEGER;
```

```
USE P2 (ARRAY[1..10] OF REAL,REAL);
```

При обработке текста пакета имя типа Q будет использовано как имя регулярного типа с вещественными элементами, а S — как имя типа REAL. Заметим, что, поскольку в пакете не задано значение по умолчанию параметра Q, тип Q должен быть либо объявлен в присоединяющем блоке, либо задан в фактических параметрах пакета.

3. В х о д н о й п а р а м е т р - и м я. Допустимым значением фактического параметра, соответствующего входному параметру-имени, является имя или уточненное имя объекта. В первом случае это имя не обязательно должно быть уже объявлено. Во втором случае оно должно быть определено в каком-либо из вложенных ранее пакетов. Пакет, имеющий входной параметр-имя, обрабатывается так, как если бы все вхождения имени формального параметра были заменены в нем на значение фактического параметра (кроме тех имен, которые переобъявлены в пакете на более глубоких блочных уровнях, т. е. в процедурах пакета). Заметим, что обычно этот вид параметра используется для задания входных параметров пакета, т. е. для указания имен тех объектов, которые должны использоваться в пакете. Однако он может быть использован и для выходных параметров пакета, т. е. для переименования определяемых в пакете объектов. Пример:

```
PACKAGE P3 (F:INOUT);
```

```
G := F (Y);
```

```
USE P3 (SIN);
```

В теле пакета вместо параметра F будет использовано имя SIN, т. е. указанный оператор присваивания будет выполняться так, как если бы он был записан как

```
G := SIN (Y);
```

4. В ы х о д н о й п а р а м е т р - и м я. Допустимым значением фактического параметра является такое имя, которое еще не определено в присоединяющем блоке. Это имя обязано быть уникальным в пределах этого блока. В тексте пакета должно быть сделано объявление некоторого объекта (константы, типа, переменной, процедуры, функции) с именем, совпадающим с именем данного фактического параметра. Обработка выходного параметра-имени производится так же, как и входного параметра-имени,

т. е. с помощью переименования. Отличие в обработке параметра вида OUT от параметра вида INOUT в том, что имя объявляемого объекта присоединяется не к контексту пакета, как обычно, а к контексту включающего блока. При этом имя, естественно, становится непосредственно доступным во включающем блоке. В теле пакета оно остается доступным, как и любое другое имя из контекста включающего блока. Такой вид параметров имеет смысл только при использовании вложенных пакетов. Для присоединенных пакетов вид OUT эквивалентен виду INOUT. Пример:

```
PACKAGE P4 (FIND:OUT);  
PROCEDURE FIND . . .
```

```
USE H=P4 (НАЙТИ);
```

Процедура FIND из пакета P4 будет непосредственно доступна во включающем блоке под именем НАЙТИ.

Аппарат выходных параметров-имен является четвертым способом доступа к объектам вложенного пакета, дополняющим возможности, предоставляемые первыми тремя способами (уточненными именами, оператором присоединения и разделом присоединения), рассмотренными в п. 3.3.

Заметим, что если параметры-константы и параметры-типы не дают никаких новых возможностей по сравнению с аппаратом ключевых параметров, то параметры-имена позволяют расширить возможности аппарата параметризации. Рассмотрим это различие на примерах.

Если снабдить уже использованный нами ранее пакет VECTSUM заголовком вида

```
PACKAGE VECTSUM (VECTOR:TYPE);
```

то включение пакета может делаться так:

```
USE VS1=VECTSUM (ARRAY[0..M] OF INTEGER);
```

Здесь позиционный параметр всего лишь заменяет ключевой.

Полиморфная процедура MAXINDEX, приведенная в п. 2.5, имеет, по сути, два параметра, имя регулярного типа VECTOR и имя функции преобразования элемента массива в сравнимый ключ. Поэтому заголовок пакета должен иметь вид

```
PACKAGE MAXINDEX (VECTOR:TYPE; KEY:INOUT);
```

При этом во время включения пакета мы сможем задать имя любой объявленной ранее функции как значение фактического параметра KEY, и именно эта функция будет использована при компиляции пакета и, естественно, при выполнении функции MAXINDEX. Например,

```

TYPE W = . . . ;
FUNCTION Q(X:ELEM(W)):INTEGER;

. . .
USE M=MAXINDEX(W,Q);

```

Заметим, что если фактическим параметром, соответствующим параметру-константе, является не константа или параметром, соответствующим параметру-типу, является не тип, то ошибка будет выявлена уже при компиляции заголовка пакета. Однако, если некорректным является значение фактического параметра, соответствующего параметру-имени, то ошибка будет выявлена только при первом использовании этого имени. Например, если мы используем следующий раздел присоединения:

```

TYPE W= . . . ;
PROCEDURE Q( . . . );

. . .
USE M=MAXINDEX(W,Q);

```

то ошибка выявится только при первом вызове в теле пакета функции KEY. Разумеется, все некорректности такого рода будут выявлены во время компиляции; так что это не повлияет на надежность программы.

### 3.6. Пакеты-функции

Еще одним способом доступа к объектам вложенного пакета является аппарат так называемых *пакетов-функций*. Введение в язык АТ-Паскаль этого средства подсказано следующим соображением: если в пакете объявляется всего лишь один объект, то использование для доступа к нему уточненного имени становится избыточным, так как уже само имя контекста однозначно идентифицирует этот объект. Поэтому в языке АТ-Паскаль действует следующее правило: если имя некоторого объявляемого в пакете объекта совпадает с именем пакета, указанным в заголовке, то для доступа к этому объекту извне пакета можно использовать само имя контекста пакета. Обычно этот аппарат применяется, когда в пакете объявляется всего один объект, но если все же таких объектов несколько, то для доступа к остальным могут использоваться обычные методы: уточненные имена, оператор присоединения и др. В этом случае имя контекста играет двойную роль, оно является одновременно и именем некоторого выделенного объекта пакета, и собственно именем контекста.

Заметим, что пакеты VECTSUM и MAXINDEX, использованные в п. 3.5, являются именно пакетами-функциями, так как в них имена объявляемых функций совпадают с именами пакетов, ука-



занными в заголовках. Поэтому пакет VECTSUM, например, можно использовать так:

```
USE SUM=VECTSUM(ARRAY[K] OF T);  
VAR S:ELEM(SUM.VECTOR);  
    V:SUM.VECTOR;  
  
S := SUM(V);
```

Имя контекста пакета SUM здесь используется и в качестве имени контекста в уточненном имени SUM.VECTOR, и в качестве имени функции VECTSUM. Термин «пакет-функция» введен по той причине, что включение такого пакета и синтаксически, и семантически напоминает вызов функции. Сравните, например, конструкции

```
USE Y=F(X) (включение пакета F с аргументом X под именем Y)
```

и

```
Y:=F(X) (присваивание переменной Y значения, вычисляемого функцией F от аргумента X).
```

Пакет-функция является аналогом вызова функции в определенной нами ранее системе операций над объектами периода компиляции.

Заметим, что пакет-функция вовсе не обязательно должен определять функцию, как могло показаться из приведенных примеров. Чтобы подтвердить это, рассмотрим в качестве примера пакет, содержащий объявление матрицы:

```
PACKAGE MATRIX(I1,I2,E:TYPE);  
TYPE I2 = I1;  
    E = REAL; ~  
VAR MATRIX:ARRAY[I1,I2] OF E;
```

Параметры пакета — это соответственно тип первого и второго индекса матрицы и тип ее элементов. Тип первого индекса является обязательным параметром. Тип второго индекса по умолчанию полагается идентичным типу первого индекса, т. е. по умолчанию матрица считается квадратной. Тип элементов матрицы по умолчанию полагается равным REAL.

Раздел присоединения такого пакета вида

```
USE A = MATRIX(1..N);
```

полностью эквивалентен такому объявлению:

```
VAR A:ARRAY[1..N,1..N] OF REAL;
```

Раздел присоединения следующего вида:

```
USE B = MATRIX(CHAR,1..2,BCOLEAN);
```

```
VAR B:ARRAY[CHAR,1..2] OF BOOLEAN;
```

Разумеется, этот пакет приведен здесь в иллюстративных целях, так как его использование не дает почти никаких удобств или экономии, однако вполне можно представить себе пакеты, содержащие только объявления переменных со сложными параметризуемыми типами.

Рассмотрим еще один пример. В программах на языке Паскаль, имеющих дело с обработкой строк, часто встречаются объявления строк, имеющие в языке Паскаль вид упакованных массивов символов. Эти объявления выглядят так:

```
VAR S:PACKED ARRAY[1..N] OF CHAR;
```

Если определить пакет-функцию следующего вида:

```
PACKAGE STR(LENGTH:CONST);
```

```
VAR STR:PACKED ARRAY[1..LENGTH] OF CHAR;
```

то объявления строк можно будет делать так:

```
USE S = STR(N);
```

```
    C = STR(20);
```

Заметим, что если мы объявим еще одну строку

```
USE T = STR(20);
```

то строки T и C окажутся разного типа. Чтобы они имели одинаковые типы, строку T надо объявлять так:

```
VAR T:TYP(C);
```

### 3.7. Процедуры с переменным числом параметров

Кроме настройки процедур пакета на обработку данных требуемых типов аппарат параметризации пакетов обладает еще одним побочным свойством. Он позволяет настраивать процедуры пакета не только на нужные типы параметров, но и на требуемое количество параметров. Процедуры пакета, количество параметров которых может варьироваться от одного использования пакета к другому, будем называть процедурами с переменным числом параметров. Разумеется, количество параметров такой процедуры, зафиксированное при присоединении пакета, в дальнейшем остается неизменным и не может изменяться при разных ее вызовах.

Аппарат создания процедур с переменным числом параметров весьма прост и основан на одном свойстве параметров-имен, состоящем в следующем. Если имя параметра объявляемой в пакете процедуры совпадает с именем параметра пакета (параметра-имени)

и значение этого параметра задано при присоединении или вложении пакета, то процедура компилируется так, как если бы данный параметр вообще отсутствовал в ее заголовке. Все обращения к этому параметру в теле процедуры тогда будут идти как обращения к объекту, объявленному в объемлющем блоке, имя которого определено значением фактического параметра пакета.

Таким образом, этот аппарат предусматривает два уровня сопоставления параметров процедуры с конкретными значениями фактических параметров: часть параметров связывается при присоединении пакета, оставшиеся «свободными» параметры связываются, как обычно, при вызове процедуры.

Рассмотрим примеры. Предположим, что у нас есть функция интегрирования, обладающая следующим заголовком:

```
FUNCTION INTEGR( FUNCTION F(X:REAL):REAL;  
    — — F — — ПОДЫНТЕГРАЛЬНАЯ  
                ФУНКЦИЯ  
    X0:REAL; — — НИЖНИЙ ПРЕДЕЛ  
    X1:REAL; — — ВЕРХНИЙ ПРЕДЕЛ  
    EPS:REAL — — ТОЧНОСТЬ  
):REAL;
```

Снабдив эту функцию заголовком пакета следующего вида:

```
PACKAGE INTEGR(F,X0,X1,EPS:INOUT);
```

и поместив полученный пакет в библиотеку, мы получаем возможность использовать эту функцию так, как показано ниже. В примерах мы будем предполагать, что в программе объявлены вещественные константы A и B, вещественные переменные C, D, E и две функции F1 и F2.

Пр и м е р ы:

1. Все параметры остаются свободными.

```
USE INT1 = INTEGR;
```

```
    . . .  
D := INT1(F1,A,C,E);
```

2. Фиксируются имя функции и точность.

```
USE INT2=INTEGR(F2,,E);
```

```
    . . .  
D := INT2(A,C);
```

3. Фиксируются пределы интегрирования и точность.

```
USE INT3 = INTEGR(,A,B,E);
```

```
    . . .  
D := INT3(F1);  
C := INT3(F2);
```

4. Фиксируются все параметры.

USE INT4 = INTEGR(F2,A,C,E):

D := INT4; — — СВОБОДНЫХ ПАРАМЕТРОВ НЕТ

Использование этого аппарата позволяет сократить списки параметров при вызове процедур и функций с большим набором параметров (а в вычислительных приложениях списки параметров могут насчитывать до десяти и более параметров). Это не только сокращает длину текста, но и, что более важно, повышает надежность программы, так как уменьшает вероятность ошибки в списках параметров многократно вызываемых процедур.

### 3.8. Локальные объекты и приватные типы

Как было сказано, область доступности объявленных во вложенном пакете имен распространяется за пределы пакета до конца включающего блока. Однако это не всегда удобно. Некоторые объекты (процедуры, переменные, константы, типы) могут иметь чисто вспомогательную роль. В этом случае доступ к ним извне пакета должен быть закрыт. Разумеется, в описании пакета можно оговорить, какими именно объектами пользоваться нельзя. Однако, где гарантия, что эти объекты действительно не будут использоваться и что, захотев изменить определение такого объекта, мы тем самым не нарушим работоспособность программы? Даже при разработке программы одним программистом он может забыть о том, что когда-то воспользовался таким вспомогательным объектом для других целей где-то вне пакета. При разработке же программы коллективом из нескольких программистов эта ситуация может еще более усугубиться. Гарантию того, что какой-либо объект используется только в теле пакета, может дать исключительно объявление этого объекта как *локального* в данном пакете. Объекты, объявленные локальными, доступны только в теле самого пакета и во включенных в него пакетах. Для того чтобы объявить объект локальным, надо перед разделом объявлений указать зарезервированное слово `LOCAL`. Например,

```
LOCAL VAR N:INTEGER;  
LOCAL CONST E = 50;  
LOCAL PROCEDURE P;
```

и т. п.

Объявленные так объекты недоступны извне пакета никакими способами, подобно тому, как недоступны извне процедуры объявленные в ней переменные.

Заметим, что в языке АТ-Паскаль принят подход, отличающийся от подхода, использованного в некоторых других языках, например в языке Модула-2 [9]. Там все объекты модуля считаются по умолчанию локальными, кроме тех, имена которых заданы в специальном списке экспортируемых имен. В языке АТ-Паскаль, напротив, все объекты по умолчанию считаются доступными извне пакета, кроме тех, которые явно объявлены локальными. Можно спорить о преимуществах и недостатках того или иного подхода. В языке АТ-Паскаль такой подход использован для того, чтобы любая совокупность объявлений (например, просто текст процедуры) считалась правильным пакетом. Это упрощает использование накопленных уже библиотек процедур на языке Паскаль как библиотек пакетов.

По аналогичным соображениям иногда бывает необходимо обеспечить гарантию того, что вне пакета, определяющего некоторый абстрактный тип, не используется знание деталей его реализации. Объявить такой тип локальным нельзя, так как имя типа должно быть доступно для использования, например для создания переменных этого типа. В этом случае тип объявляется *приватным* (термин языка Ада, где он означает почти то же самое). Перед объявлением приватного типа указывается зарезервированное слово PRIVATE. В теле пакета и во всех включенных в него пакетах сохраняются обычные правила доступа к нему и его использования. Однако вне пакета имя приватного типа можно использовать только для объявления переменных. Приватный тип не может быть аргументом деструктора типа. Переменные приватных типов можно использовать только в операторах присваивания и как фактические параметры вызовов процедур и функций.

Пр и м е р.

```
PACKAGE COMPLEX;
PRIVATE TYPE COMPLEX =
    RECORD
        RE, IM: REAL
    END;
```

. . .

Во включающем блоке можно выполнять такие действия:

```
USE COMPLEX;
VAR A,B,C: COMPLEX;
FUNCTION F(X,Y: COMPLEX): COMPLEX;
```

. . .

```
A := B;
C := F(A,B);
```

но нельзя такие:

```
A.RE := 0;
B.IM := C.IM;
```

### 3.9. Ограничение видимости

Ограничение видимости пакета позволяет четко зафиксировать список объявленных во внешних контекстах имен, которые используются в пакете. Это достигается включением перед заголовком пакета специальной конструкции вида

RESTRICTED (*список имен*)

Если список имен опущен, то никакие имена из объемлющих контекстов недоступны в пакете, кроме предопределенных имен (стандартных функций, процедур и т. п.).

Наличие ограничения видимости позволяет сконцентрировать всю информацию об интерфейсе пакета с «внешним миром» в его заголовке, так что при необходимости изменить представление какого-либо объекта всегда можно с гарантией сказать, какие пакеты или процедуры используют его имя, а какие нет. Без наличия ограничения видимости установление этого факта могло бы потребовать просмотра всего текста пакета. Имена позиционных параметров пакета считаются неявно присутствующими в списке видимости пакета, так что повторять их нет необходимости. Ограничение видимости применимо к отдельным процедурам точно так же, как и к целым пакетам.

### 3.10. Пакеты как процедуры периода компиляции

В главе 2 уже отмечалось, что компиляцию исходного текста программы следует рассматривать как протекающий во времени процесс, в котором выполняются разнообразные действия над объектами-именами и объектами-типами. Попробуем развить этот подход, чтобы показать глубокие аналогии, существующие между средствами работы с абстрактными типами данных и общеалгоритмическими средствами языка программирования. Общепринятый подход к процессу компиляции состоит в том, что ее понимают как процесс преобразования исходного текста программы на некотором языке в эквивалентную программу на языке более низкого уровня. Обычно — это машинные команды некоторой реальной или абстрактной машины. С точки зрения конечного результата компиляции это так. Но, рассматривая подробно именно процесс компиляции, мы увидим, что не меньшее, если не большее значение, чем генерация объектного кода, имеют операции над внутренними структурами данных компилятора. Эти структуры, назовем их внутренней базой данных компилятора, состоят из объектов-имен и объектов-типов, связанных системой ссылок в графовые структуры большой сложности. Причем, если генерируемая последовательность машинных команд зависит не только от исходной программы, но и от

системы команд машины, то состояния внутренней базы данных зависят только от исходного текста компилируемой программы, прямо отражая ее структуру. Сгенерированные однажды машинные команды обычно уже не обрабатываются больше компилятором (за исключением случая оптимизации объектного кода), в то время как внутренняя база не только заполняется, но и интенсивно используется в течение всего процесса компиляции. Блок в языке Паскаль состоит из двух синтаксически и семантически различающихся частей: разделов объявлений и собственно тела блока. При обработке разделов объявлений производится заполнение внутренней базы, при обработке же тела она только используется, за исключением создания новых контекстных уровней в операторе присоединения. Каждое появление в тексте компилируемой программы некоторого имени приводит к обращению к базе данных компилятора, причем от текущего состояния этой базы в большой степени зависит, какая именно последовательность машинных команд будет сгенерирована. Особенно хорошо это видно на примере полиморфных процедур.

Таким образом, можно сказать, что если тело блока при компиляции преобразуется в вид, готовый для выполнения каким-либо иным, отличным от компилятора процессором, физическим или программным, то разделы объявлений блока «выполняются» (интерпретируются) самим компилятором. Разумеется, в некоторых случаях, например при обработке раздела инициализации переменных, компилятор также может строить исполняемый код. Однако в данном случае этим можно пренебречь. Мы будем далее говорить о «выполнении» объявлений именно в смысле «выполнения» их компилятором.

Рассмотрим теперь аналогию между процессом выполнения тела блока и процессом выполнения объявлений. Аналогом переменных периода компиляции являются объекты-имена. Аналогом значений переменных являются ссылки на объекты-типы. Аналогом операции присваивания — объявление имени. Как уже отмечалось в главе 2, конструкторы и деструкторы типов можно трактовать как вызовы стандартных функций. При таком подходе включение пакета оказывается не чем иным, как вызовом процедуры. Так же, как обычная процедура при выполнении использует значения своих входных параметров или глобальных переменных и присваивает вычисленные ею значения выходным параметрам или глобальным переменным, пакет использует объявленные ранее имена (т. е. использует значения объектов-имен) и объявляет другие объекты (т. е. создает объекты-имена и присваивает им значения — ссылки на объекты-типы). Как и процедуру, пакет можно «вызывать» несколько раз с различными значениями параметров, при этом и результаты вызова будут различными. Так же, как возвращаемое

значение функции задается в языке Паскаль присваиванием значения имени функции, значение пакета-функции определяется объявлением объекта с тем же именем, что и пакет, т. е. все тем же присваиванием значения объекту-имени. Пакет-функция, объявляющий несколько объектов, т. е. присваивающий значения нескольким объектам-именам, соответствует функции с побочными эффектами.

Остановимся еще на возможности передать пакету через параметры имя другого пакета, который должен быть использован. Рассмотрим пакет следующего вида:

```
PACKAGE P(Q:INOUT);  
USE S = Q;
```

Здесь параметр Q используется в разделе присоединения как внешнее имя пакета. Включение такого пакета (или, как мы можем теперь говорить, *вызов* его) может быть сделано так:

```
USE P(R);
```

При обработке пакета P будет вызван пакет R и использованы значения, присваиваемые им объектам-именам. Это — аналог передачи имени процедуры как фактического параметра при вызове обычной процедуры. Этот пример еще раз подчеркивает аналогию между процедурами периода выполнения и пакетами, понимаемыми как процедуры периода компиляции.

В языке ПЛ/1 также используется термин *функции периода компиляции*. Чтобы устранить возможные ассоциации, отметим, что так называемые средства периода компиляции языка ПЛ/1 на самом деле действуют не в период собственно компиляции, а на этапе работы препроцессора, который предварительно обрабатывает исходный текст, поступающий затем на вход компилятора.

Препроцессор ПЛ/1 есть не что иное, как специализированный макропроцессор. Таким образом, функции периода компиляции ПЛ/1 и описанные средства АТ-Паскаль имеют принципиальное различие, так как действуют на разных этапах обработки исходного текста и над разными объектами. Первые — до начала компиляции над исходным текстом, вторые — непосредственно во время компиляции над внутренней базой данных компилятора. Хотя, конечно, некоторые простые возможности средств АТ-Паскаль (например, присоединение пакетов без параметров) могут имитироваться средствами периода компиляции ПЛ/1 или аналогичными макросредствами других языков программирования. В то же время препроцессор ПЛ/1 обладает такими средствами, которые отсутствуют в АТ-Паскаль, например средствами условной генерации.



## ГЛАВА 4

# ОСНОВНЫЕ ПРИНЦИПЫ АТД-ПРОГРАММИРОВАНИЯ

### 4.1. Классификация пакетов

В предыдущих главах мы рассмотрели языковые средства языка АТ-Паскаль, обеспечивающие возможность АТД-программирования. Перейдем теперь к вопросам применения этих средств, в частности к определению с их помощью абстрактных типов данных. Как должно быть ясно из предшествующих глав, возможности аппарата параметризуемых пакетов и средств декомпозиции типов гораздо шире, чем только определение АТД. Можно выделить по крайней мере три класса пакетов:

- 1) пакеты, содержащие произвольные совокупности объявлений и используемые для обеспечения модульности и повышения надежности при традиционных способах программирования;
- 2) пакеты, определяющие АТД (будем называть их АТД-пакетами);
- 3) пакеты, определяющие полиморфные процедуры и функции (назовем их ПП-пакетами).

Примеры всех трех классов пакетов будут рассмотрены ниже.

Хотя используемые при создании АТД-пакетов и ПП-пакетов языковые средства совпадают, эти классы пакетов различаются следующим. Входными параметрами АТД-пакета обычно являются константы и типы. Выходными параметрами АТД-пакета является обычно имя определяемого АТД и имена операций над объектами этого типа. При этом АТД-пакет может использовать пакеты, определяющие другие АТД, используемые при определении данного. Таким образом, этот тип пакета выполняет функцию синтеза более сложного типа из простых (АТД или предопределенных). Заметим, что здесь и далее в этой главе мы используем понятие *параметр пакета* в обобщенном смысле. Это не обязательно должен быть параметр, определяемый в заголовке пакета. Здесь под входными

параметрами пакета мы понимаем любые имена используемых в нем объектов, а псд выходными параметрами — имена определяемых им объектов.

ПП-пакет, напротив, получает в качестве входного параметра обычно некоторый сложный тип и, возможно, имена операций над ним. ПП-пакет декомпозирует сложный тип на составные части и, используя их, конкретизирует некоторый заданный в нем алгоритм, настраивая его на обработку данных этого типа.

Таким образом, можно сказать, что АТД-пакеты предназначены для определения абстрактных данных, а ПП-пакеты — для определения абстрактных алгоритмов.

По широте области применения пакеты можно разделить на два больших класса: *стандартные* и *пользовательские*. Стандартные АТД-пакеты реализуют некоторую согласованную совокупность понятий, которая может рассматриваться как расширение базового языка и применяться в самых разнообразных программах разными программистами. Примерами таких пакетов могут служить пакеты для работы с комплексными числами, векторами и матрицами, строками переменной длины, списками, деревьями, таблицами и т. п. Пользовательские АТД-пакеты рассматриваются как средство иерархической структуризации программы. Они разрабатываются для использования в одной программе и, как правило, не имеют самостоятельного значения вне ее. Мы видим здесь все ту же аналогию между пакетами и процедурами, которые также разделяются на применимые в большом числе разнообразных программ и разрабатываемые для одной программы. Аналогичное разделение существует и у ПП-пакетов. Стандартные ПП-пакеты представляют собой наиболее употребительные алгоритмы в самом общем виде, позволяющие после надлежащей конкретизации применять их в самых разнообразных программах. Пример — алгоритм сортировки. Применение таких ПП-пакетов позволяет сократить время разработки программы за счет отсутствия необходимости в ручной привязке известного алгоритма к разрабатываемой программе. Одновременно повышается и надежность программ, так как при ручной привязке неизбежны ошибки, которые удается обнаружить далеко не сразу. Сокращаются и затраты на тестирование программ. Применение пользовательских ПП-пакетов целесообразно в том случае, когда один и тот же алгоритм должен применяться в различных местах программы к данным разных типов.

## 4.2. Основные свойства АТД-программирования

1. **М о д у л ь н о с т ь.** Основным видом модуля является пакет. В общем случае пакет определяет некоторый АТД, т. е. в нем определяется тип и операции над объектами этого типа. В частных

случаях пакет может содержать одну, возможно, полиморфную процедуру или произвольные совокупности объявлений констант, типов и переменных.

2. Недоступность реализации АТД. В АТД-пакете должны четко разделяться две его стороны:

— интерфейс пакета, т. е. совокупность используемых и определяемых им имен, и спецификация свойств объектов, обозначаемых этими именами (типы параметров процедур и др.);

— реализация пакета, т. е. представление АТД через более простые типы, алгоритмы операций над ним, локальные типы, переменные и процедуры.

С помощью соответствующих языковых средств реализация пакета может быть сделана недоступной для использования в том блоке, где используется пакет. Блоки, использующие АТД-пакет, не должны использовать знание деталей реализации АТД.

3. Взаимозаменяемость реализаций. Строгое следование этому принципу должно обеспечить возможность замены реализации пакета, например, на более эффективную, без каких бы то ни было изменений использующей пакет программы.

4. Многослойная структура программы. Программа, состоящая из множества АТД-пакетов, разбивается на слои так, что каждый  $n$ -й слой обеспечивает определение АТД, используемых в пакетах  $(n-1)$ -го слоя, и в свою очередь использует АТД, определенные в пакетах  $(n+1)$ -го слоя. На верхнем уровне этой пирамиды стоит главная программа, на нижних — АТД, определяемые только через предопределенные типы.

5. Максимальное использование стандартных пакетов. При создании программы следует стремиться к максимальному использованию ранее определенных АТД, расширяющих возможности базового языка. Некоторые из таких АТД определены в самой реализации языка АТ-Паскаль. Они описаны ниже.

6. Структурность процедур пакетов. Тела процедур пакетов должны создаваться с использованием известных принципов структурного программирования. Как отмечалось в главе 1, АТД-программирование является не заменой, а дальнейшим развитием структурного программирования, поэтому его применение позволяет использовать весь накопленный объем знаний о принципах проектирования хорошо структурированных программ. Более того, АТД-программирование позволяет более четко сформулировать некоторые правила структурного программирования, давая им языковую поддержку. Например, в структурном программировании постулируется необходимость разбиения программ на

модули небольшого размера, выполняющие вполне определенные, четко специфицированные функции. Оговаривается, какие управляющие структуры можно использовать для программирования модулей, но, как правило, решение вопроса о том, как наилучшим образом разбить программу на модули, остается целиком за программистом. Даже в наиболее фундаментальных монографиях, посвященных структурному программированию, часто приводятся только общие рекомендации следующего вида: «Основная идея ... пошагового совершенствования состоит в разбиении заданной функции на некоторые части (подфункции) с последующим объединением и проверкой этих частей, так, что исходная функция выражается посредством эквивалентной структуры соответствующим образом соединенных подфункций» [10]. Концепция АТД-программирования отвечает на вопрос, как разбивать «заданную функцию» на «подфункции» (термин «функция» здесь употребляется, конечно же, не в смысле языка Паскаль, а в более общем). Разбивать надо так, чтобы каждая подфункция определяла некоторый АТД, а заданная функция выражалась через операции над этими типами в виде одной простой и понятной процедуры небольшого объема.

**7. Применение метода пошаговой реорганизации.** Аппарат АТД-программирования обеспечивает языковую поддержку так называемым стратегиям пошагового совершенствования и пошаговой реорганизации программ (см., например, [10]). Поскольку эти стратегии представляются в наилучшей степени согласующимися с принципами АТД-программирования, рассмотрим кратко (уже в терминах АТД) стратегию пошаговой реорганизации, в которую пошаговое совершенствование входит как составная часть. «Эта стратегия основана на стремлении сохранить простоту восприятия сложной программы путем первоначального программирования с помощью пошагового совершенствования (без учета эффективности программы) и последующего программирования посредством пошаговой реорганизации программы с целью построения более эффективной программы» [10].

Первый шаг построения программы состоит в том, что мы, рассматривая всю задачу в целом, пытаемся определить те АТД, в терминах операций над которыми эта задача решалась бы достаточно просто. Определив интерфейс пакетов, реализующих эти АТД, мы можем запрограммировать первый слой программы и даже выполнить пробную компиляцию с целью выявления некорректности использования операций над АТД. На следующем шаге мы берем в качестве исходной задачи реализацию одного из пакетов второго слоя. Для нее определяется представление реализуемого АТД и программируется реализация операций над ним. Возможно, при этом потребуются определить АТД третьего слоя. Определяется интерфейс пакетов третьего слоя и программируются процедуры па-

кета второго слоя. Этот шаг повторяется для всех пакетов второго, третьего и последующих слоев до тех пор, пока не останется больше нереализованных АТД. При этом целесообразно, конечно, совмещать программирование некоторого пакета с более или менее формальным доказательством того, что реализация процедур пакета правильно отражает свойства операций над АТД. Способы формальных доказательств правильности программ описываются, например, в [11]. Основной целью на этом этапе является построение правильной и хорошо структурированной программы. При выборе методов реализации АТД следует использовать методы как можно более простые, забыв о соображениях эффективности. Результатом первого этапа является работающая отлаженная программа, которую можно считать как бы первым приближением к конечной цели, натурной моделью или прототипом окончательного варианта программы. Может оказаться, что эта программа уже удовлетворяет исходным требованиям к эффективности. Тогда можно будет посчитать это первое приближение последним и на этом остановиться. Чаще, конечно, она не будет удовлетворять поставленным требованиям (или чувство профессиональной гордости программиста не даст ему оставить свою программу в таком «сыром» неэффективном варианте). Тогда начинается следующий этап разработки — собственно пошаговая реорганизация.

На втором этапе определяются те АТД, реализация которых в наибольшей степени отвечает за общую неэффективность программы. Для этого могут быть использованы специальные средства построения профилей работы программ, измеряющие, сколько процентов потребляемого программой процессорного времени уходит на выполнение той или иной процедуры. На каждом шаге этого этапа выбирается один пакет и его реализация изменяется так, чтобы достичь большей эффективности (заметьте, большей, но не наибольшей). При этом в силу принципа взаимозаменяемости реализаций, сформулированного выше, замена реализации пакета  $n$ -го слоя не требует никаких переделок в пакетах  $(n-1)$ -го слоя, кроме, конечно, того случая, когда по каким-либо причинам требуется изменить интерфейс пакетов. При такой замене реализации, поскольку уже есть работающий прототип пакета и вся программа в целом, довольно легко протестировать новую версию как автономно, так и в комплексе. После каждого шага реорганизации делаются замеры эффективности программы. Этот процесс продолжается до тех пор, пока эффективность программы не достигнет желаемой степени. Может оказаться, что для одних наборов исходных данных или для одних режимов эксплуатации программы более эффективна одна реализация некоторого пакета, а для других — другая. В этом случае почти без всяких дополнительных затрат может быть сгенерировано несколько версий одной и той же

программы, различающихся только реализацией некоторых АТД. Для этого достаточно перекомпилировать программу, задав использование той или иной библиотеки пакетов.

### 4.3. Об эффективности АТД-программирования

Используя термин *эффективность программы*, мы понимаем под ним затраты процессорного времени и оперативной памяти, требуемые для ее выполнения. Описывая выше применение стратегии пошаговой реорганизации к построению АТД-программ (будем называть так программы, построенные в строгом соответствии с принципами АТД-программирования), мы не ставили и, естественно, не отвечали на вопрос: «А что же делать, если никакие пошаговые реорганизации не позволяют достичь нужной степени эффективности программы, и вообще может ли АТД-программа быть столь же эффективной, как программа на том же языке, но написанная в традиционном стиле?» Оставив пока без ответа первую часть вопроса, попробуем ответить на вторую. За повышение надежности, легкости понимания и модификации программы, конечно же, приходится расплачиваться некоторой потерей эффективности, но ... А далее укажем несколько возможных «но».

1. В настоящее время пространственно-временные характеристики программ все более отодвигаются на второй план по сравнению с такими характеристиками, как надежность, легкость сопровождения и модификации, познаваемость. Это связано в первую очередь с быстрым ростом технических характеристик современных компьютеров. Программа, не удовлетворяющая заданным критериям временной эффективности, может оказаться вполне приемлемой при выполнении ее на другом, более мощном процессоре. Появление машин с виртуальной памятью вообще практически устраняет использование объема памяти в качестве критерия сравнения программ. То есть критерий эффективности программ по времени и памяти на поверку часто оказывается критерием соответствия аппаратной части ЭВМ требованиям данной задачи. Этот критерий относителен, в то время как критерий надежности абсолютен. Никакое совершенствование технических средств не заставит надежно работать программу, содержащую ошибки.

2. Аналогичную ситуацию мы наблюдаем и в других областях техники. Никого ведь не удручает, что надежный пассажирский лайнер не может поставить рекорд высоты, скорости или дальности полета. Да и в самом программировании все, например, уже примирились с тем, что языки высокого уровня, более легкие в использовании и обеспечивающие больший уровень надежности, дают менее эффективную программу, чем языки низкого уровня.

3. Потери от снижения эффективности программ всегда следует соизмерять с потерями от усложнения процесса отладки и модификации, от того, что «высокоэффективную» чужую программу часто невозможно понять и приходится переписывать ее заново, чтобы внести в нее какие-либо изменения.

4. Средства АТД-программирования могут быть реализованы так, чтобы минимизировать дополнительные затраты при выполнении АТД-программы. Например, само по себе использование параметризуемых пакетов, деструкторов типов, уточненных имен, оператора присоединения и других средств управления видимостью имен в языке АТ-Паскаль никак не сказывается на эффективности программы. Все эти языковые средства обрабатываются на этапе компиляции и не требуют дополнительных затрат при выполнении. Более того, в некоторых случаях использование аппарата АТД-программирования позволяет даже улучшить эффективность программы. Например, аппарат пакетов языка АТ-Паскаль позволяет производить статическую настройку процедуры на обработку тех или иных переменных вызывающей программы. Тем самым при ее вызове сокращается количество передаваемых параметров и снижаются затраты. Можно привести и другой пример. Если переменная объявлена так:

```
VAR J:INDEX(A);
```

то при выполнении оператора

```
FOR J DO A[J] := B;
```

можно без потери надежности отключить динамическую проверку выхода индекса за границы массива. Здесь значение переменной J не может выйти за пределы массива по определению такого вида оператора цикла.

Но, разумеется, в общем случае применение методики АТД-программирования ухудшает эффективность программы за счет того, что для доступа к деталям реализации АТД приходится использовать вызовы процедур и функций.

Что же касается первой части поставленного вопроса, то, как нам кажется, ответ на него уже дан. В качестве же крайней меры для повышения эффективности программы можно в наиболее критических модулях, выявленных с помощью построения профилей выполнения, использовать такой прием, как *раскрытие представления АТД*. При этом мы снимаем запрет на использование знания деталей реализации АТД. Этот прием аналогичен тому, как в отлаженной программе на языке высокого уровня заменяют наиболее критические модули аналогичными модулями на языке низкого уровня. Например, если в некоторой программе используется тип COMPLEX для работы с комплексными числами, объявленный так:

```

TYPE COMPLEX =
  RECORD
    RE,IM:REAL
  END;

```

то при строгом следовании принципам АТД-программирования для доступа к вещественной и мнимой частям некоторой комплексной переменной, объявленной так:

```
VAR Z:COMPLEX;
```

надо использовать обращение к функциям  $RE(Z)$  и  $IM(Z)$ . Знание представления типа, вообще говоря, позволяет делать такие обращения более эффективно —  $Z.RE$  и  $Z.IM$ . Следует, однако, помнить, что выполнять такое *раскрытие типа* надо в самом крайнем случае и только на последних этапах пошаговой реорганизации программы, но ни в коем случае не на ранних. Если сделать это на ранних этапах, нарушится принцип взаимозаменяемости реализаций АТД. Если, например, на каком-то шаге обнаружится, что более эффективным было бы использование тригонометрического представления комплексных чисел, сделать такую замену реализаций будет уже очень трудно. Скорее всего, такая замена совсем не будет сделана, и мы, желая улучшить эффективность с помощью раскрытия типа на раннем этапе разработки, на самом деле ухудшим ее.

Разумеется, во многих случаях раскрытие представления АТД могло бы быть сделано самим компилятором. В этих случаях операторы вызова процедур и функций из АТД-пакета должны компилироваться не в обычный объектный код вызова процедуры с передачей параметров и выделением памяти под локальные переменные, но непосредственно в код, соответствующий телу блока процедуры со статической подстановкой параметров. Поскольку это раскрытие представления АТД выполняет компилятор, оно остается недоступным для программиста и качество программы не ухудшается. Эффективность же АТД-программ при этом может значительно повыситься. Однако в существующем компиляторе языка АТ-Паскаль такая возможность отсутствует,

#### 4.4. Способы объявления объектов АТД

Существуют два основных способа построения АТД-пакетов, различающиеся универсальностью и надежностью. В первом случае в пакете объявляется тип, обычно приватный, имя которого и используется во включающем блоке для объявления переменных. Этот способ наиболее универсален, так как позволяет использовать произвольное количество переменных и массивов данного типа,



делать объекты АДТ полями записей, объявлять функции, возвращающие значения данного типа. Во втором случае тип, объявляемый в пакете, локализован в нем. Кроме объявления типа в пакете объявляется и переменная этого типа (или массив с компонентами этого типа). Все процедуры и функции пакета оперируют с этой

```
PACKAGE STACK( T:TYPE; M:CONST );
```

```
PRIVATE TYPE STACK =
```

```
    RECORD
```

```
        L:0 .. M;
```

```
        V:ARRAY[1 .. M] OF T
```

```
    END;
```

```
FUNCTION TOP( S:STACK ):T;
```

```
    BEGIN TOP := S.V[S.L] END;
```

```
    . .
```

а) Универсальный способ

```
PACKAGE STACK( T:TYPE; M:CONST );
```

```
LOCAL TYPE STACK = . . .
```

```
LOCAL VAR S:STACK;
```

```
FUNCTION TOP:T;
```

```
    BEGIN TOP := S.V[S.L] END;
```

```
    . . .
```

б) Внутреннее представление объекта

```
PACKAGE STACKARRAY( IND,T:TYPE; M:CONST );
```

```
LOCAL TYPE STACK = . . .
```

```
LOCAL VAR S:ARRAY[ IND ] OF STACK;
```

```
FUNCTION TOP( J:IND );
```

```
    BEGIN WITH S[J] DO TOP := V[L] END;
```

```
    . . .
```

в) Внутреннее представление массива объектов

Рис. 4. Способы построения АДТ-пакетов

переменной или с элементами массива, индексы которых передаются им через параметры. Если бы выше мы не подвергли критике критерий «копеечной» эффективности, можно было бы сказать, что этот способ эффективнее, так как сокращает количество передаваемых процедурам параметров. Этот способ более надежен, так как

доступа к локальной переменной извне пакета нет. На рис. 4 приведены фрагменты трех пакетов, определяющих АД *стек*. *Стек* представлен в виде записи, первое поле которой указывает на текущую вершину стека, а второе — массив, хранящий помещенные в стек значения, которые могут быть произвольного типа. Над стеком могут быть определены следующие операции: инициализация стека, помещение нового значения в вершину стека, выборка значения из вершины стека с выводением его из стека, выборка значения с сохранением его в стеке (функция TOP) и получение глубины стека, т. е. количества помещенных в него значений.

Рассмотрим, как следует обращаться к функции TOP в этих трех вариантах пакета (переменная D объявлена как символьная).

#### 1. Присоединение пакетов,

```
a) USE STACK(CHAR,20);
   VAR S:STACK;
```

```
       . . .
   D := TOP(S);
```

```
b) USE STACKVAR(CHAR,20);
```

```
       . . .
   D := TOP;
```

```
в) USE STACKARRAY((A,B,C),CHAR,20);
```

```
       . . .
   D := TOP(A);
```

#### 2. Вложение пакетов.

```
a) USE CSTACK = STACK(CHAR,20);
   VAR S:CSTACK;
```

```
       . . .
   D := CSTACK.TOP(S);
```

```
b) USE CSTACK = STACKVAR(CHAR,20);
```

```
       . . .
   D := CSTACK.TOP;
```

```
в) USE CSTACK = STACKARRAY((A,B,C),CHAR,20);
```

```
       . . .
   D := CSTACK.TOP(A);
```

В случае присоединения второй пакет проигрывает в наглядности, так как отсутствует прямая текстуальная связь между именем функции и пакетом. Для того чтобы определить, в каком из используемых пакетов определена функция TOP (и функция ли это или переменная), пользователь должен будет просмотреть тексты или описания всех используемых пакетов. В случае вложения пакетов первый способ кажется чересчур громоздким, а наиболее естественным представляется второй. В нем имя пакета выступает в роли имени объекта, следом за которым записывается применяе-

мая к нему операция. Однако предпочтение, отдаваемое тому или иному виду использования пакета, в общем-то, дело вкуса.

В чем же принципиальное отличие этих способов построения и использования пакетов? Они отличаются по глубине «упрятывания» информации о представлении АТД. В пакете универсального вида имя типа доступно пользователю. Несмотря на то, что тип объявлен приватным, тем не менее существуют способы «взлома» его представления, такие как записи с вариантами и функции преобразования типа. Если программист, использующий пакет, сознательно подчиняется жесткой дисциплине АТД-программирования, то, конечно, все способы равно безопасны. Однако каждый опытный программист знает, как велик соблазн улучшить эффективность программы любой ценой. И если он где-то в сложной программе «взламывает» абстрактный тип, т. е. непосредственно использует, пусть даже вполне корректно, знание его представления, то велика вероятность того, что через год, захотев изменить реализацию типа, он об этом «взломе» не вспомнит. Последствия читатель себе, видимо, представляет.

Второй способ построения пакета исключает возможность «взлома», так как ни имя типа, ни имя переменной в программе недоступны. Заметим, что получить доступ к типу нельзя никакими деструкторами типов, так как их применение к приватным типам запрещено. Применение способа построения пакетов с внутренним представлением объекта, конечно же, более ограничено, чем универсального. Например, трудно представить себе пакет работы с комплексными числами, все процедуры которого работают с одной-единственной комплексной переменной. В примерах, приводимых далее, мы будем в основном придерживаться универсальной структуры пакета,

## ГЛАВА 5

### ПРИМЕРЫ ПАКЕТОВ

#### 5.1. Простые пакеты

Из определения пакета следует, что любая совокупность объявлений, помещенная в библиотеку пакетов, может рассматриваться как пакет и включаться в программу во время компиляции при указании ее имени в разделе присоединения. Это может быть текст отдельной процедуры, совокупность объявлений констант, типов, переменных или все вместе. Допускается также использование пакета, состоящего только из одних комментариев. В случае процедур аппарат параметризации пакетов частично устраняет неудобство, связанное с отсутствием в языке Паскаль параметров-массивов переменного размера (в международном стандарте языка Паскаль такая возможность есть — так называемые конформные массивы (conformant arrays), но в языке АТ-Паскаль и ряде других реализаций эта возможность не реализована). Наличие аппарата параметризации пакетов неизбежно подталкивает нас к тому, чтобы делать любую разрабатываемую процедуру по возможности полиморфной.

Предположим, что в некоторой программе используется процедура присвоения одного и того же значения всем элементам некоторого массива:

```
TYPE VECTOR = ARRAY[1..N] OF INTEGER;  
  
PROCEDURE ASSIGN (VAR A:VECTOR;B:INTEGER);  
    VAR J:1..N;  
BEGIN  
    FOR J DO A[J] := B  
END;
```

Если подобная процедура часто используется в различных программах, возникает желание поместить ее в библиотеку пакетов, чтобы не переписывать каждый раз заново. При этом тип VECTOR

естественно сделать параметром пакета. Поскольку при составлении пакета этот тип заранее не известен, параметр В следует объявить с типом ELEM(VECTOR), а индексную переменную J — с типом INDEX(VECTOR). Пакет принимает следующий вид:

```
PACKAGE ASSIGN (VECTOR:TYPE);
PROCEDURE ASSIGN (VAR A:VECTOR;B:ELEM(VECTOR));
    VAR J:INDEX (VECTOR);
BEGIN
    FOR J DO A[J] := B
END;
```

Теперь эта процедура стала полиморфной, она пригодна для работы с массивами, имеющими произвольные типы индексов и элементов:

```
TYPE INTARR = ARRAY[0..M] OF INTEGER;
TYPE CHRARR = ARRAY[CHAR] OF CHAR;
USE ASSINT = ASSIGN(INTARR);
USE ASSCHR = ASSIGN(CHRARR);
VAR K:INTARR;
    C:CHRARR;

    . . .
ASSINT(K,0);

    . . .
ASSCHR(C,'*');
```

Если процедура вызывается многократно в одной и той же программе для присвоения значений элементам одного и того же массива, ее вызов можно сделать более удобным и эффективным, настраивая процедуру на имя нужного массива и нужное присваиваемое значение при присоединении пакета, а не при вызове. При этом вызов процедуры будет производиться без параметров, Вид пакета будет таким:

```
PACKAGE ASSIGN(A:INOUT;B:CONST);
PROCEDURE ASSIGN;
    VAR J:INDEX(A)
BEGIN
    FOR J DO A[J] := B
END;
```

Пример использования пакета:

```
TYPE INTARR = ARRAY[0..M] OF INTEGER;
TYPE CHRARR = ARRAY[CHAR] OF CHAR;
VAR K:INTARR;
    C:CHRARR;
```

```
USE ASSK = ASSIGN(K,0);  
USE ASSC = ASSIGN(C,'*');
```

ASSK; — — ПРИСВАИВАЕТ 0 ЭЛЕМЕНТАМ МАССИВА K  
ASSC; — — ПРИСВАИВАЕТ '\*' ЭЛЕМЕНТАМ МАССИВА C

Заметим, что если присваиваемое значение должно быть не константой, а вычисляемым значением, то параметр В в заголовке пакета надо объявить как параметр-имя (с видом INOUT) и при присоединении пакета поставить ему в соответствие имя той переменной, значение которой будет присваиваться элементам массива.

В качестве второго примера тривиального применения пакетов рассмотрим случай использования отдельно компилируемых процедур (называемых в языке Паскаль *внешними* процедурами). Известно, что последние являются потенциальным источником ненадежности программ, проистекающей из-за того, что при обработке вызова внешней процедуры компилятор не может проверить, полностью ли соответствует список фактических параметров вызова внешней процедуры списку ее формальных параметров. Хотя в вызывающем блоке любую использованную внешнюю процедуру надо объявить, задав ее список параметров, однако полной корректности это не гарантирует. Дело в том, что это объявление внешней процедуры в использующей ее программе пишет не разработчик процедуры, а разработчик вызывающей программы, следовательно, возможны случайные ошибки и опечатки. Кроме того, в языке АТ-Паскаль допускается использование внешней процедурой глобальных переменных, объявленных в главной программе. Они должны быть объявлены в модуле компиляции внешней процедуры, причем в том же порядке, что и в главной программе, так как сопоставление глобальных переменных, объявленных в главной программе и во внешней процедуре, проводится не по именам, а по порядку следования, как в блоке COMMON языка Фортран. Часто также в различных внешних процедурах должны использоваться одни и те же константы и типы. Для увеличения надежности и легкости модификации программ с внешними процедурами можно применить следующую технику. Все объявления глобальных констант, переменных и типов объединяются в пакеты. Для каждой внешней процедуры (или для набора процедур, используемых всегда совместно) также заводится пакет, содержащий ее (их) объявление. Поскольку это объявление помещается в библиотеку пакетов всего один раз и делает это сам разработчик процедуры, здесь можно достичь полного соответствия. Присоединение этого пакета в вызывающей программе обеспечивает полную гарантию корректности интерфейса с внешней процедурой. Предположим, что пользователь занес в библиотеку следующие пакеты:

- 1) PACKAGE GLOBCONST;  
CONST N = 120;  
D = '\*';
- 2) PACKAGE GLOBVAR;  
VAR A,B,C:REAL;  
M:INTEGER;
- 3) PACKAGE EXTPROC;  
PROCEDURE EXTPROC(L:INTEGER;  
VAR R:REAL); PASCAL;

Тогда вызывающая программа будет выглядеть так:

```
PROGRAM EXAMPLE(OUTPUT);
USE GLOBCONST,
    GLOBVAR,
    EXTPROC;
VAR N:INTEGER;
    F:REAL;
    . . .
EXTPROC(N,F);
    . . .
```

Модуль компиляции внешней процедуры будет иметь следующий вид:

```
PROGRAM EXAMPLE(OUTPUT);
USE GLOBCONST,
    GLOBVAR;
PROCEDURE EXTPROC(L:INTEGER;
    VAR R:REAL);
    BEGIN
    . . .
    END;
```

## 5.2. Пакеты для обработки строк переменной длины

Рассмотрим теперь настоящий АД-пакет, который обеспечивает все необходимые операции для работы с абстрактным типом данных *строка переменной длины*. Известно, что одним из недостатков языка Паскаль является отсутствие удобных средств работы со строками, подобных тем, какие есть, например, в языке Г.Л/1. Заметим, что в некоторых диалектах языка Паскаль эти средства введены, например, в системе UCSD PASCAL [12], применяющейся на персональных компьютерах фирмы IBM. Там эти средства вводятся путем модификации самого языка и компилятора. Мы же воспользуемся для этой цели универсальным аппаратом определения АД языка АТ-Паскаль. Рассмотренный ниже пакет может

служить прототипом стандартного пакета, обеспечивающего расширение возможностей базового языка.

Рассмотрим сначала интерфейс пакета, т. е. набор операций, определенных над объектами типа *строка*. Дадим этому типу имя STRING. В качестве одного из возможных наборов операций взяты операции, в основном совпадающие с теми, которые используются в системе UCSD PASCAL. Пакет STRING будет содержать определение типов STRING и DISPL (переменные типа DISPL хранят позиции в строке, они могут принимать значения от 0 до максимальной длины строки), а также следующие процедуры и функции:

1) FUNCTION CONCAT(A,B:STRING):STRING;

Значение функции есть строка, полученная сцеплением строк A и B.

2) FUNCTION COPY(A:STRING;L,N:DISPL):STRING;

Значение функции — подстрока строки A длиной N, начинающаяся с L-й позиции строки A.

3) FUNCTION POS(B,A:STRING):DISPL;

Значение функции — номер позиции, начиная с которой строка B входит в строку A как подстрока. Если такой подстроки в A нет, то значение функции есть нуль.

4) PROCEDURE DELETE(VAR A:STRING;L,N:DISPL);

Удаление из строки A подстроки длиной N, начинающейся с L-й позиции.

5) PROCEDURE INSERT(B:STRING;

VAR A:STRING;L:DISPL);

Вставка строки B в строку A после позиции с номером L,

6) FUNCTION LENGTH(A:STRING):DISPL;

Значение функции — текущая длина строки A.

7) FUNCTION STRFIX(A:STRING)

:PACKED ARRAY[1..MAXLENGTH] OF CHAR;

Значение функции — строка фиксированной длины, равной максимальной длине строк переменной длины (MAXLENGTH).

8) FUNCTION STRVAR(A:PACKED ARRAY[1..MAXLENGTH]  
OF CHAR):STRING;

Функция преобразует строку фиксированной длины (MAXLENGTH) в строку переменной длины с текущей длиной, равной максимальной.

9) FUNCTION STRUNC(A:STRING):STRING;

Функция усекает все пробелы в конце строки.

10) FUNCTION EMPTY:STRING;

Значение функции есть пустая строка.

11) FUNCTION REP(N:DISPL;S:CHAR):STRING;

Значение функции — строка, состоящая из N символов S,



Заметим, что в спецификации пакета отсутствует группа операций, часто необходимая в определении АТД, а именно операции сравнения. Можно было бы определить такие функции, возвращающие значения типа BOOLEAN, однако сравнение строк переменной длины можно делать, сначала преобразовав их в строки фиксированной длины, а затем выполняя сравнение обычными средствами языка Паскаль. Пример:

```
VAR X,Y:STRING;
```

```
IF STRFIX(X) = STRFIX(Y) THEN . . .
```

Отсутствует и процедура для присваивания значения переменной типа STRING. Такая процедура не нужна, так как в языке Паскаль операция присваивания определена над любыми типами,

```
PROCEDURE REPLACE( VAR S:STRING; P,R:STRING);
```

```
VAR J:DISPL; -- ПОЗИЦИЯ ПОДСТРОКИ P
```

```
L:DISPL; -- ДЛИНА ПОДСТРОКИ P
```

```
BEGIN
```

```
J := POS(P,S);
```

```
L := LENGTH(P);
```

```
WHILE J > 0 DO -- ПОКА ЕСТЬ ВХОЖДЕНИЯ P В S
```

```
BEGIN
```

```
DELETE(S,J,L); -- УДАЛИТЬ ПОДСТРОКУ P
```

```
INSERT(R,S,J-1); -- ВСТАВИТЬ НА ЕЕ МЕСТО R
```

```
J := POS(P,S) -- НАЙТИ СЛЕДУЮЩЕЕ ВХОЖДЕНИЕ
```

```
END
```

```
END;
```

Рис. 5. Процедура замены подстрок в строке переменной длины

Но, конечно, предполагая, что над переменными типа STRING допустима операция присваивания, мы тем самым накладываем ограничение на возможные способы реализации АТД. Предположение, что объект типа STRING будет представлен одной переменной, допустимо не при всякой реализации.

Не говоря пока ничего о реализации АТД, рассмотрим пример, показывающий, как используются операции над АТД и как пользователь может определить свои процедуры и функции над объектами типа STRING. Рассмотрим процедуру REPLACE (рис. 5), заменяющую все вхождения подстроки P в строке S на строку R. Из подобных процедур пользователь при необходимости может сформировать свой пакет работы со строками, расширяющий набор операций стандартного пакета. Мы убедились в том, что тип STRING действительно представляет собой абстрактный тип. Мы

построили использующую этот тип процедуру, ничего не зная о реализации типа.

Рассмотрим теперь реализацию пакета STRING. Здесь возможно несколько альтернативных вариантов. Наиболее простым представлением типа STRING является запись вида

```
RECORD  
  LENGTH:0..MAXLENGTH;  
  VAR:PACKED ARRAY[1..MAXLENGTH] OF CHAR  
END;
```

Поле LENGTH хранит текущую длину строки, не превышающую заранее заданного ограничения MAXLENGTH. Первые LENGTH элементов массива VAL представляют текущее значение строки. Достоинством такого представления является его простота и легкость записи алгоритмов процедур пакета. Недостаток такого способа представления в том, что необходимо заранее фиксировать максимальную длину, причем строки с разными максимальными длинами будут иметь неидентичные типы. Полный текст пакета приведен на рис. 6. Обратите внимание на то, каким образом присваивается возвращаемое значение структурной функции. Доступ к полям возвращаемого значения производится в точности так же, как и доступ к полям записи. Процедура INSERT намеренно написана с использованием других процедур пакета, хотя более эффективной была бы ее непосредственная реализация, такая же как у других процедур. На рис. 7 представлен еще один вспомогательный пакет STRINGIO, содержащий процедуры ввода-вывода строк переменной длины. Они выделены в отдельный пакет, так как, видимо, не всякая программа, работающая со строками переменной длины, будет в них нуждаться. Если пакет STRING в программе присоединяется, то он определяет тип STRING, который непосредственно используется в пакете STRINGIO. Если же в программе используется несколько типов, порождаемых пакетом STRING, с разными максимальными длинами, надо при использовании пакета STRINGIO передать ему имя типа через его первый параметр. Например,

```
USE S1 = STRING(20),  
    S2 = STRING(80),  
    IO1 = STRINGIO(S1),  
    IO2 = STRINGIO(S2);
```

В этом случае обращаться к процедурам пакетов надо с использованием аппарата косвенного доступа, например,

```
VAR P,Q: S1;  
    F1:TEXT;  
    IO1.STROUTPUT(F1,S1.CONCAT(P,Q));
```

```

-- ПАКЕТ ДЛЯ РАБОТЫ СО СТРОКАМИ ПЕРЕМЕННОЙ ДЛИНЫ
--   MAXLENGTH - МАКСИМАЛЬНАЯ ДЛИНА СТРОК
--   ВЫХОДНЫЕ "ПАРАМЕТРЫ" (ОПРЕДЕЛЯЕМЫЕ ИМЕНА)
--   STRING - ТИП СТРОКИ ПЕРЕМЕННОЙ ДЛИНЫ
--   DISPL - ПОЗИЦИЯ В СТРОКЕ
--   FIXSTRING - СТРОКА ФИКСИРОВАННОЙ ДЛИНЫ MAXLENGTH
-- ПАКЕТ ОПРЕДЕЛЯЕТ ПРОЦЕДУРЫ И ФУНКЦИИ:
--   CONCAT, COPY, POS, DELETE, INSERT, LENGTH, STRFIX,
--   STRVAR, STRUNC, EMPTY, REP, ELEMENT

PACKAGE STRING( MAXLENGTH:CONST );
  CONST MAXLENGTH = 80; -- ДЛИНА ПО УМОЛЧАНИЮ
  TYPE DISPL = 0..MAXLENGTH;
  STRING =
    RECORD
      LENGTH:DISPL;
      VAL:PACKED ARRAY[1..MAXLENGTH] OF CHAR
    END;
  FIXSTRING = PACKED ARRAY[1..MAXLENGTH] OF CHAR;

-- НАЗНАЧЕНИЕ ПРОЦЕДУР ОПИСАНО В ГЛАВЕ 5
FUNCTION CONCAT( A,B:STRING ):STRING;
  VAR N,K:DISPL;
  BEGIN
    FOR K := 1 TO A.LENGTH DO
      CONCAT.VAL[K] := A.VAL[K]
    N := A.LENGTH;
    FOR K := 1 TO B.LENGTH DO
      IF N < MAXLENGTH THEN BEGIN
        N := N+1;
        CONCAT.VAL[N] := B.VAL[K]
      END;
    CONCAT.LENGTH := N
  END;

```

Рис. 6. Пакет для работы со строками переменной длины

```

FUNCTION COPY( A:STRING; L,N:DISPL ):STRING;
  VAR M,K:DISPL; J:INTEGER;
  BEGIN
    M := 0;
    FOR K := 1 TO N DO
      BEGIN
        J := L+K-1;
        IF J <= MAXLENGTH THEN
          BEGIN
            COPY.VAL[K] := A.VAL[J];
            M := M+1
          END
        END;
      COPY.LENGTH := M
    END;
  END;

```

```

FUNCTION POS( B,A:STRING ):DISPL;
  LABEL 1,2;
  VAR J,L,K,N:INTEGER;
  COMP:BOOLEAN;
  BEGIN
    L := A.LENGTH-B.LENGTH+1;
    IF L >= 1 THEN
      BEGIN
        FOR K := 1 TO L DO
          BEGIN
            COMP := FALSE;
            FOR J := 1 TO B.LENGTH DO
              IF A.VAL[K+J-1]<>B.VAL[J]
                THEN GOTO 1;
            COMP := TRUE;
          END
        END;
      END;
    END;
  END;

```

Рис. 6. Пакет для работы со строками переменной длины

```

1: IF COMP THEN
    BEGIN
        POS := K; GOTO 2
    END
    ELSE POS := 0;
END
END
END
ELSE POS := 0;
2:END;

PROCEDURE INSERT( B:STRING; VAR A:STRING; L:DISPL );
BEGIN
    A := CONCAT(
        COPY(A,1,L),
        CONCAT(B,COPY(A,L+1,MAXLENGTH))
    )

END;

FUNCTION LENGTH( A:STRING ):DISPL;
BEGIN LENGTH := A.LENGTH END;

FUNCTION STRFIX( A:STRING ):TYP(String.VAL);
VAR K:DISPL;
BEGIN
    STRFIX := A.VAL;
    FOR K := A.LENGTH+1 TO MAXLENGTH DO
        STRFIX[K] := ' '
    END;
END;

FUNCTION STRVAR( A:FIXSTRING ):STRING;
BEGIN
    STRVAR.VAL := A;
    STRVAR.LENGTH := MAXLENGTH
END;

```

Рис. 6. Пакет для работы со строками переменной длины

```

FUNCTION STRUNC( A:STRING ):STRING;
BEGIN
    WITH A DO
        WHILE ( VAL[LENGTH] = ' ' )
            AND ( LENGTH > 1 ) DO
            LENGTH := LENGTH-1;
        STRUNC := A
    END;

FUNCTION EMPTY:STRING;
BEGIN EMPTY.LENGTH := 0 END;

FUNCTION REP( N:DISPL; S:CHAR ):STRING;
VAR K:DISPL;
BEGIN
    FOR K := 1 TO N DO REP.VAL[K] := S;
    REP.LENGTH := N
END;

-- КОНЕЦ ТЕКСТА ПАКЕТА STRING

```

Рис. 6. Пакет для работы со строками переменной длины

Заметим, что имена входного и выходного файлов (INF и OUTF) являются как параметрами пакета, так и параметрами процедур. Это означает, что процедуры имеют списки параметров переменной длины. Имя файла может быть указано в параметре пакета, тогда указывать его в параметрах каждого конкретного вызова процедур пакета не надо. Если же в параметре пакета имя файла не задано, то процедуры пакета могут работать с разными текстовыми файлами, но имена файлов надо включать в список параметров каждого вызова.

Чтобы завершить рассмотрение средств работы со строками переменной длины, нам осталось решить еще одну небольшую задачу. У нас отсутствуют средства задания констант типа STRING. При малой максимальной длине строки для этой цели можно воспользоваться функцией STRVAR, преобразующей строку фиксированной длины в строку с переменной длиной. Например,

```

USE STRING(8);
VAR S:STRING;

. . .
S := STRVAR('ABCDEFGH');

```

```

-- ПАКЕТ ВВОДА-ВЫВОДА СТРОК ПЕРЕМЕННОЙ ДЛИНЫ
-- ВХОДНЫЕ "ПАРАМЕТРЫ"
--   STRING - ТИП СТРОК ПЕРЕМЕННОЙ ДЛИНЫ
--           (ОПРЕДЕЛЯЕТСЯ В ПАКЕТЕ STRING)
--   INF     - ИМЯ ВХОДНОГО ФАЙЛА
--   OUTF    - ИМЯ ВЫХОДНОГО ФАЙЛА
-- ВЫХОДНЫЕ "ПАРАМЕТРЫ" ПАКЕТА (ОПРЕДЕЛЯЕМЫЕ ИМЕНА)
--   STRINPUT - ПРОЦЕДУРА ВВОДА ИЗ ФАЙЛА INF СТРОКИ,
--               ОГРАНИЧЕННОЙ С ДВУХ СТОРОН АПОСТРОФАМИ.
--               ЕСЛИ ДЛИНА ПРЕВЫШАЕТ МАКСИМАЛЬНУЮ,
--               СТРОКА УСЕКАЕТСЯ.
--   STROUTPUT - ВЫВОД СТРОКИ В ФАЙЛ OUTF.
--               ВЫВОДИТСЯ СТОЛЬКО СИМВОЛОВ,
--               КАКОВА ТЕКУЩАЯ ДЛИНА СТРОКИ.
--   STRINPLINE - ВВОД ОЧЕРЕДНОЙ СТРОКИ ТЕКСТОВОГО ФАЙЛА
--               INF С ПРЕОБРАЗОВАНИЕМ ЕЕ В СТРОКУ
--               ПЕРЕМЕННОЙ ДЛИНЫ

```

```

PACKAGE STRINGIO( STRING:TYPE; INF,OUTF:INOUT );
PROCEDURE STRINPUT( VAR INF:TEXT; VAR S:STRING );
  VAR C:CHAR;
      J:INTEGER;
BEGIN
  J := 0;
  REPEAT READ(INF,C) UNTIL C = ''';
  LOOP
    IF EOLN(INF) THEN EXIT;
    READ(INF,C);
    IF C = '''' THEN
      BEGIN
        READ(INF,C); IF C <> '''' THEN EXIT
      END;

```

Рис. 7. Пакет ввода-вывода строк переменной длины

```

        IF J < LAST(INDEX(STRING.VAL)) THEN
        BEGIN
            J := J+1;
            S.VAL[J] := C
        END
    END (* LOOP *);
    S.LENGTH := J
END;

PROCEDURE STROUTPUT( VAR OUTF:TEXT; VAR S:STRING );
BEGIN
    IF S.LENGTH > 0 THEN WRITE(OUTF,S.VAL:S.LENGTH)
END;

PROCEDURE STRINPLINE( VAR INF:TEXT; VAR S:STRING );
BEGIN
    S.LENGTH := LINELENGTH(INF);
    READLN(INF,S.VAL)
END;

```

- КОНЕЦ ТЕКСТА ПАКЕТА STRINGIO

Рис. 7. Пакет ввода-вывода строк переменной длины

Однако при больших длинах строк этот способ становится неудобным, так как фактический параметр функции STRVAR должен иметь длину, в точности равную максимальной длине строки (MAX-LENGTH). Поэтому удобно было бы иметь пакет, преобразующий строковую константу любой длины в строку переменной длины. Пример такого пакета показан на рис. 8. После использования такого пакета следующего вида:

```

USE STRING(100),
C1 = STRCONST(STRING, 'ABCD'),
C2 = STRCONST(STRING, 'EFGH');

```

имена C1 и C2 становятся как бы именами констант 'ABCD' и 'EFGH' типа STRING и могут использоваться в выражениях

```
VAR S:STRING;
```

```

    S := C1;           — S = 'ABCD'
    S := CONCAT(S,C2); — S = 'ABCDEFGH'

```

Если у нас есть несколько типов строк переменной длины с разными максимальными длинами, то определяемые с помощью пакета



STRCONST «константы» привязываются к тому или иному типу с помощью первого параметра пакета — имени типа,

```
USE S1 = STRING(20),
    S2 = STRING(80),
    C1 = STRCONST(S1, 'ABCD'),
    C2 = STRCONST(S2.STRING, 'EFGH');
```

Так как пакет STRING является пакетом-функцией, имя контекста пакета может использоваться непосредственно как имя определяемого типа, поэтому в параметрах пакета STRCONST выражения S2 и S2.STRING эквивалентны.

```
-- ПАКЕТ ДЛЯ ОПРЕДЕЛЕНИЯ КОНСТАНТ ТИПА STRING
-- ИСПОЛЬЗОВАНИЕ:
--   USE <ИМЯ КОНСТАНТЫ>=STRCONST(<ИМЯ ТИПА>,<СТРОКА>);
-- ЕСЛИ <ИМЯ ТИПА> ЕСТЬ STRING, ЕГО МОЖНО ОПУСКАТЬ
-- НАПРИМЕР:
--   USE Q = STRCONST(STR,'ТЕКСТ ПРОИЗВОЛЬНОЙ ДЛИНЫ');
```

```
PACKAGE STRCONST( STRING:TYPE; C:CONST )
FUNCTION STRCONST:STRING;
    TYPE V = TYP( STRING.VAL );
    VAR P:TYP(C);
    BEGIN
        P := C;
        STRCONST.LENGTH := SIZE(P);
        STRCONST.VAL := V(P) -- V ИСПОЛЬЗУЕТСЯ
                               -- КАК ФУНКЦИЯ
                               -- ПРЕОБРАЗОВАНИЯ ТИПА
    END;
```

Рис. 8. Пакет для определения констант типа STRING

На самом деле, как видно из текста пакета, имена C1 и C2 обозначают не константы, а функции типа STRING, возвращающие в качестве своих значений константы, заданные как параметры пакета STRCONST.

Возможны разные варианты построения пакетов для обработки строк переменной длины. Мы рассмотрели всего лишь один из них. Даже в нем возможны некоторые удобные модификации. Например, начало пакета STRINGIO могло бы выглядеть так:

```
PACKAGE STRINGIO(MAXLENGTH:CONST;
                 INF,OUTF:INOUT);
```

USE STRING;  
— — ДАЛЕЕ, КАК НА РИС. 7

...

В этом случае не было бы необходимости присоединять пакеты STRING и STRINGIO по отдельности. Использование пакета STRINGIO автоматически повлекло бы за собой присоединение пакета STRING. Заметим, что передавать константу MAXLENGTH как первый параметр пакета STRING нет необходимости, раз она задана в заголовке пакета STRINGIO. Такой способ построения пакета STRINGIO удобнее для использования, однако он не позволяет при присоединении пакета зафиксировать имена нескольких входных и выходных файлов, как это возможно при присоединении нескольких пакетов STRINGIO для одного типа STRING. Например,

```
VAR FILE1,FILE2:TEXT;  
USE STRING(80);  
USE F1 = STRINGIO(FILE1),  
    F2 = STRINGIO(FILE2);  
VAR S:STRING;
```

```
    . . .  
F1.STRIPLINE(S);
```

```
    . . .  
F2.STRIPLINE(S);
```

### 5.3. Пакет для работы с комплексными числами

Средства работы с комплексными числами отсутствуют как в стандартной версии языка Паскаль, так и практически во всех известных реализациях, хотя в некоторых приложениях вычислительного характера они могут оказаться нужными. Однако пакет для работы с комплексными числами мы рассмотрим в первую очередь не из-за его полезности, а из-за того, что на его примере хорошо демонстрируется принцип взаимозаменяемости реализаций абстрактного типа. Мы рассмотрим только принципы построения пакета, оставив разработку полного текста в качестве упражнения для читателя.

Пакет должен определить тип COMPLEX и набор операций над ним, к которым, видимо, целесообразно отнести следующие:

1. Арифметические операции — функции CADD (сложение), CSUB (вычитание), CMLT (умножение), CDIV (деление), CPWR (возведение в степень).

2. Функции выделения действительной и мнимой частей комплексного числа и вычисления его модуля — RE, IM, CABS.

3. Стандартные функции CSIN, CCOS, CEXP, CLOG, CSQRT и др.

4. Процедуры ввода и вывода комплексных чисел CREAD и CWRITE.

5. Функция формирования комплексного числа из двух вещественных — CMPLX.

Рассмотрим вопрос о выборе представления комплексных чисел. Как известно, в математике приняты две основные формы представления комплексных чисел: алгебраическое представление вида  $X+iY$  и тригонометрическое —  $\rho(\cos \varphi + i \sin \varphi)$ . И в том, и в другом случае комплексное число можно представить записью, состоящей из двух полей. Различие только в том, как трактуются эти поля, т. е. в записи алгоритмов операций над ними. Если нас совершенно не волнует эффективность, можно выбрать любое представление. Но если критерий эффективности все же играет роль, то вопрос о выборе того или иного представления не решается однозначно. Одни операции, например сложение и вычитание, реализуются более эффективно над алгебраической формой представления, другие, например умножение, деление и возведение в степень, — над тригонометрической. Конечно же, можно составить два отдельных пакета и использовать в программе тот или иной в зависимости от специфики данной программы. Однако существует и другой подход, который лучше отражает возможность взаимозаменяемости реализаций АТД. Он состоит в том, что создается один пакет, содержащий только объявление типа COMPLEX и интерфейсы операций над ним. Имя же пакета, содержащего реализацию процедур, задается как параметр этого пакета.

```
PACKAGE COMPLEX(IMPLEMENTATION:INOUT);  
  PRIVATE TYPE COMPLEX =  
    RECORD  
      X, Y:REAL  
    END;  
  FUNCTION CADD(A,B:COMPLEX):COMPLEX;  
    FORWARD;  
  FUNCTION CSUB(A,B:COMPLEX):COMPLEX;  
    FORWARD;  
  . . .  
  FUNCTION CMPLX(R,S:REAL):COMPLEX;  
    FORWARD;  
  USE IMPLEMENTATION;
```

Напомним, что директива FORWARD, записанная вместо тела процедуры или функции, в языке Паскаль означает, что тело процедуры или функции будет дано ниже по тексту программы,

Пакет, реализующий алгебраическое представление комплексных чисел, будет иметь вид

```
PACKAGE REIM;  
FUNCTION CADD(* (A,B:COMPLEX):COMPLEX *);  
  BEGIN  
    CADD.X := A.X + B.X;  
    CADD.Y := A.Y + B.Y  
  END;  
  . . . и т. д.
```

Аналогично строится и другой пакет реализации (назовем его RHOPHI), содержащий реализацию тригонометрической формы представления комплексных чисел.

```
PACKAGE RHOPHI;  
FUNCTION CMLT(* (A,B:COMPLEX):COMPLEX *);  
  BEGIN  
    CMLT.X := A.X * B.X;  
    CMLT.Y := A.Y + B.Y  
  END;  
  . . . и т. д.
```

Теперь, если мы хотим использовать в программе комплексные числа с той или иной формой представления, соответствующие разделы присоединения будут иметь вид

```
USE COMPLEX(REIM);
```

или

```
USE COMPLEX(RHOPHI);
```

Здесь мы видим, как внешнее имя пакета может быть передано другому пакету как параметр. При этом сам текст программы никак не зависит от выбранной формы представления. Все операции над комплексными числами будут выражаться только через обращения к функциям пакета. Например, так:

```
VAR U,V,W:COMPLEX;  
    R,Q:REAL;  
    . . .  
U := CMPLX(1.5, -2.0);  
V := CMPLX(0.2, 5.7);  
W := CADD(U, CMLT(V, CMPLX(R, Q)));
```

Пакеты, содержащие реализацию АТД, кроме тел функций и процедур могут содержать и наиболее употребительные константы, выраженные в виде структурных констант языка АТ-Паскаль, например, действительную и мнимую единицу.

CONST

ONE = (# 1.0,0.0 #);

IONE = (# 0.0,1.0 #);

Если требуется, чтобы обе формы представления могли использоваться в одной и той же программе, необходимо добавить еще функции преобразования комплексных чисел из одной формы представления в другую.

Заметим, что в этом примере мы существенно использовали тот факт, что обе формы представления комплексных чисел используются запись, состоящую из двух вещественных полей. Если бы это было не так, то нам пришлось бы все-таки сделать два различных пакета работы с комплексными числами, хотя и с одинаковыми интерфейсами. Можно сказать, что описанный прием применим только тогда, когда представление АТД в памяти остается неизменным, а меняются только алгоритмы выполнения операций над ним.

#### 5.4. Обработка списков

Пакет обработки списков LIST, рассматриваемый ниже, обеспечивает набор простых средств для манипулирования динамическими списковыми структурами. Техника обработки списков в настоящее время хорошо известна, так что мы не будем останавливаться на деталях. К сожалению, многие приемы обработки списков ориентированы именно на детали реализации, т. е. основаны на знании того, как представлены элементы списков и как устроены связи между ними. Мы постараемся выбрать такую концептуальную модель, которая была бы наиболее абстрактной и по возможности надежной. Известно, что использование адресных ссылок, необходимое при построении программ, работающих со списками, является одним из источников трудно обнаруживаемых ошибок.

Воспользуемся следующей концептуальной моделью. *Список* есть упорядоченное множество элементов. Каждый элемент списка характеризуется тремя атрибутами: ссылкой на предыдущий элемент списка, ссылкой на следующий элемент и значением. Тип значений элементов списка может быть произвольным, исключая файловый. Все элементы одного списка должны иметь один и тот же тип. Каждый элемент может быть включен только в один список, однако разные элементы одного и того же или разных списков могут иметь одинаковые значения. Элемент, не имеющий предыдущего, называется первым элементом списка. Элемент, не имеющий последующего, называется последним.

Пакет LIST, определяющий средства работы со списками, имеет один основной параметр — тип значений, хранимых в элементах списка. Его имя VALTYPE. Пакет, в отличие от ранее рассмотренных, определяет не один, а два типа — тип LIST и тип ELEMENT. Объекты, принадлежащие типу LIST, представляют

сами списки. Объекты, принадлежащие типу ELEMENT,— элементы списков.

Над объектами типа LIST определены следующие операции:

1) FUNCTION NEWLIST:LIST;

Создать новый пустой список.

2) FUNCTION FIRSTEL(S:LIST):ELEMENT;

Взять первый элемент списка.

3) FUNCTION LASTEL(S:LIST):ELEMENT;

Взять последний элемент списка.

4) PROCEDURE INCLUDE(N:VALTYPE;S:LIST);

Включить в список новый элемент с заданным значением. При выполнении процедуры создается новый элемент со значением N и включается в список, заданный вторым параметром. Если S есть элемент (типы LIST и ELEMENT идентичны, что допускает сопоставление параметру процедуры типа LIST фактического параметра типа ELEMENT), то новый элемент включается перед элементом S в тот список, которому принадлежит S. Если S есть список, то новый элемент добавляется в конец списка.

5) PROCEDURE COPYLIST(P:LIST;VAR Q:LIST);

Скопировать список, т. е. построить новый список Q, который был бы точной копией списка P. Эта процедура задает операцию присваивания над списками.

6) PROCEDURE DESTROY(VAR P:LIST);

Уничтожить список и все его элементы.

Над элементами списка определены следующие операции:

1) FUNCTION NEXTEL(X:ELEMENT):ELEMENT;

Взять элемент списка, следующий за элементом X. Если элемент X — последний, то значение функции равно NIL.

2) FUNCTION PREVEL(X:ELEMENT):ELEMENT;

Взять элемент списка, предыдущий по отношению к элементу X. Если X есть первый элемент, то значение функции равно NIL.

3) FUNCTION VALUE(X:ELEMENT):VALTYPE;

Взять значение элемента.

4) FUNCTION REMOVED(X:ELEMENT):VALTYPE;

Взять значение элемента с одновременным выведением его из списка и уничтожением.

Чтобы пояснить использование некоторых операций, рассмотрим несколько простых примеров. Будем предполагать, что в блоке есть объявления следующего вида:

USE LIST(INTEGER);

VAR V,W:LIST;

X:ELEMENT;  
N:INTEGER;

Примеры использования процедур пакета LIST:

1. Создать новый список:

V := NEWLIST;

2. Включить значение 2 в список V:

INCLUDE(2,V);

3. Включить значение N в начало списка V (перед первым элементом):

INCLUDE(N,FIRSTEL(V));

4. Удалить последний элемент списка V, присвоив его значение переменной N:

N := REMOVED(LASTEL(V));

5. Перевести элемент X из списка V в список W:

INCLUDE(REMOVED(X),W);

6. Взять значение предпоследнего элемента списка:

N := VALUE(PREVEL(LASTEL(W)));

Заметим, что единственное предположение, которое мы сделали, состоит в том, что ELEMENT — это ссылочный тип. Это следует из того, что значением функции типа ELEMENT может быть значение NIL. От типа VALTYPE определенные нами операции не требуют никаких особых свойств. Это может быть любой тип, над которым определена операция присваивания (т. е. любой тип, кроме файлового). Это позволяет строить списки числовых значений, строк, массивов, записей и т. п. Можно строить также списки списков, что позволяет обрабатывать древовидные структуры данных, однако это нецелесообразно. Для работы с древовидными структурами было бы удобно построить специализированный пакет или набор пакетов.

Можно было бы существенно расширить возможности данного пакета, добавив к нему функцию, обратную функции VALUE, т. е. такую, которая бы по заданному значению находила в списке элемент с таким значением. Однако введение такой функции, расширив возможности пакета, сузило бы множество возможных типов значений элементов. Для поиска элемента с заданным значением требуется, чтобы над значениями была определена операция сравнения на равенство. Видимо, включать такую функцию в пакет нецелесообразно. Ее можно оформить как отдельный пакет и использовать вместе с пакетом LIST тогда, когда это необходимо и когда тип значений элементов позволяет это.

```

-- ПАКЕТ ОБРАБОТКИ СПИСКОВ
-- ОПИСАНИЕ ПАКЕТА ПРИВЕДЕНО В ГЛАВЕ 5

PACKAGE LIST( VALTYPE:TYPE );
TYPE ELEMENT = @ELSTRUCT;
LIST      = ELEMENT;
ELSTRUCT=
    RECORD
        NEXT, PREV: ELEMENT;
        V: VALTYPE
    END;

-- НАД ОБЪЕКТАМИ ТИПА LIST ОПРЕДЕЛЕНЫ ОПЕРАЦИИ:
FUNCTION NEWLIST: LIST; FORWARD;
    -- СОЗДАТЬ НОВЫЙ ПУСТОЙ СПИСОК.
FUNCTION FIRSTEL( S: LIST ): ELEMENT; FORWARD;
    -- ВЗЯТЬ ПЕРВЫЙ ЭЛЕМЕНТ СПИСКА.
FUNCTION LASTEL( S: LIST ): ELEMENT; FORWARD;
    -- ВЗЯТЬ ПОСЛЕДНИЙ ЭЛЕМЕНТ СПИСКА.
PROCEDURE INCLUDE( N: VALTYPE; S: LIST ); FORWARD;
    -- ВКЛЮЧИТЬ В СПИСОК НОВЫЙ ЭЛЕМЕНТ
    -- С ЗАДАНЫМ ЗНАЧЕНИЕМ.
PROCEDURE COPYLIST( P: LIST; VAR Q: LIST ); FORWARD;
    -- СКОПИРОВАТЬ СПИСОК P В СПИСОК Q
PROCEDURE DESTROY( VAR P: LIST ); FORWARD;
    -- УНИЧТОЖИТЬ СПИСОК И ВСЕ ЕГО ЭЛЕМЕНТЫ.

-- НАД ЭЛЕМЕНТАМИ СПИСКА ОПРЕДЕЛЕНЫ ОПЕРАЦИИ:
FUNCTION NEXTEL( X: ELEMENT ): ELEMENT; FORWARD
    -- ВЗЯТЬ ЭЛЕМЕНТ СПИСКА,
    -- СЛЕДУЮЩИЙ ЗА ЭЛЕМЕНТОМ X.
FUNCTION PREVEL( X: ELEMENT ): ELEMENT; FORWARD;
    -- ВЗЯТЬ ЭЛЕМЕНТ СПИСКА,
    -- ПРЕДЫДУЩИЙ ПО ОТНОШЕНИЮ К X

```

Рис. 9. Пакет обработки списков



```

FUNCTION VALUE( X:ELEMENT ):VALTYPE;FORWARD;
    -- ВЗЯТЬ ЗНАЧЕНИЕ ЭЛЕМЕНТА.
FUNCTION REMOVED( X:ELEMENT ):VALTYPE;FORWARD;
    -- ВЗЯТЬ ЗНАЧЕНИЕ ЭЛЕМЕНТА И ВЫВЕСТИ ЕГО ИЗ СПИСКА
--$L-    ПРИ КОМПИЛЯЦИИ ТЕКСТ РЕАЛИЗАЦИИ НЕ ПЕЧАТАЕТСЯ
-- НИЖЕ ПРИВОДИТСЯ ЧАСТЬ РЕАЛИЗАЦИИ ПРОЦЕДУР ПАКЕТА

FUNCTION NEWLIST;
    VAR L:LIST;
    BEGIN
        NEW(L); -- В АТ-ПАСКАЛЬ ИСПОЛЬЗОВАТЬ ЯВНЫЙ СИМВЛ
                -- УКАЗАТЕЛЯ @ НЕОБЯЗАТЕЛЬНО
        L.NEXT := L; L.PREV := L;
        NEWLIST := L
    END;
. . .
PROCEDURE INCLUDE (* N:VALTYPE; S:LIST *);
    VAR X:ELEMENT;
    BEGIN
        S := S.PREV;
        NEW(X);
        WITH X@ DO
            BEGIN
                V := N;
                NEXT := S.NEXT;
                PREV := S
            END;
        S.NEXT.PREV := X;
        S.NEXT := X
    END;

```

Рис. 9. Пакет обработки списков

На рис. 9 приведены текст интерфейса пакета LIST и часть его реализации. Заметим, что такое текстуальное отделение интерфейса от реализации удобно и может применяться почти всегда. С помощью специальной операции компилятора можно заблокировать распечатку реализации в листинге пакета так, чтобы листинг содержал только описание интерфейса, нужное пользователю пакета. Типы LIST и ELEMENT в пакете представлены как ссылочные типы, ссылающиеся на запись с тремя полями, представляю-

щими три атрибута элемента. Поскольку типы объявлены как приватные, такие действия над переменными типов LIST и ELEMENT, как создание и уничтожение процедурами NEW и DISPOSE или доступ к полям записи, не разрешены. Из рис. 9 видно, что типы LIST и ELEMENT объявлены как идентичные. Разные имена типов введены исключительно из соображений удобства использования пакета. Именно в силу этой идентичности при обращении к процедуре включения в список нового элемента вторым параметром ее может быть как ссылка на список, так и ссылка на элемент. Заметим, что, хотя, как известно, при реализации циклических

#### -- ПОИСК В СПИСКЕ ЭЛЕМЕНТА С ЗАДАННЫМ ЗНАЧЕНИЕМ

```
PACKAGE FOUND( N,S:INOUT );
FUNCTION FOUND(N:VALTYPE; S:LIST ):ELEMENT;
  VAR X:ELEMENT;
  BEGIN
    X :=FIRSTEL(S);
    LOOP
      IF X = NIL THEN EXIT;
      IF VALUE(X) = N THEN EXIT;
      X := NEXTEL(X)
    END;
    FOUND := X
  END;
```

Рис. 10. Пакет FOUND

двунаправленных списков удобно, когда структура записи заголовка списка совпадает со структурой элемента, вопрос об идентичности типов LIST и ELEMENT мог бы быть решен и по-другому. В случае неидентичности этих типов потребовались бы две отдельные процедуры включения в список — включения в начало или конец списка и включения перед или после некоторого элемента. Зато компилятор имел бы возможность более строго контролировать соответствие типов параметров процедур. В нашем случае мы пожертвовали строгостью и частично надежностью ради простоты.

На рис. 10 приведен текст пакета, содержащего функцию FOUND для поиска в списке элемента с заданным значением. Эта процедура может служить примером использования процедур и функций пакета LIST. Рассмотрим на этом примере одну проблему, которая часто встречается при программировании двух разных, но взаимосвязанных пакетов. Она состоит в том, что при построении пакета на рис. 10 мы предполагали, что он погружен в контекст

пакета LIST, так как в нем используются имена процедур из пакета LIST. Если же пакет LIST не присоединен к объемлющему блоку, а вложен в него, то использовать определенный так пакет FOUND нельзя. Возможным решением было бы включение имени контекста пакета LIST в список параметров пакета FOUND, начало которого имело бы вид

```
PACKAGE FOUND(C:INOUT);
USE CONTEXT(C);
```

. . .

Использование пакетов в этом случае должно производиться так:

```
USE L = LIST(CHAR);
F = FOUND(L);
```

Заметим здесь, что по правилам обработки раздела присоединения контекст пакета либо присоединяется к контексту объемлю-

```
FUNCTION SETAND( A,B:LIST ):LIST;
VAR R:LIST;
    E:ELEMENT;
USE FOUND(,B);
BEGIN
    R := NEWLIST;
    E := FIRSTEL(A);
    WHILE E <> NIL DO
        BEGIN
            IF FOUND(VALUE(E)) <> NIL THEN
                INCLUDE( VALUE(E),R);
            E := NEXTEL(E)
        END;
    SETAND := R
END;
```

Рис. 11. Функция вычисления пересечения множеств, заданных списками

щего блока, либо получает собственное имя. Если по каким-то причинам необходимо и то, и другое, это можно сделать так:

```
USE L = LIST(CHAR),CONTEXT(L);
```

Действие функции FOUND можно было бы распространить на любые типы, если передавать пакету имя функции, преобразующей любое значение элемента списка в значение такого типа, над которым допускалось бы выполнение операции сравнения на равенство. Если эта функция будет преобразовывать значения эле-

ментов в значения типа BOOLEAN, то она будет не чем иным, как правилом отбора среди элементов списка такого, который удовлетворяет заданному критерию. При этом поиск может производиться, разумеется, по правилам, более сложным, чем простое сравнение.

Одной из возможных модификаций описанного выше пакета LIST могло бы быть переопределение операции INCLUDE так, чтобы она не допускала дублирования в списке одинаковых значений. В этом случае пакет предоставлял бы набор средств для выполнения операций над множествами более общего, чем множества языка Паскаль, вида. Конечно, при этом его следовало бы дополнить такими операциями, как пересечение и объединение множеств. Пример подобной процедуры, реализующей операцию пересечения множеств, представленных списками, приведен на рис. 11, как еще одна иллюстрация использования средств пакета LIST.

### 5.5. Пример АТД-программы

Чтобы построить простой, но достаточно интересный пример полной программы, использующей АТД, рассмотрим следующую задачу. Даны текстовый файл и множество пар строк  $(S_1, R_1)$ ,  $(S_2, R_2)$ , ...,  $(S_n, R_n)$ . Необходимо во всех строках исходного файла заменить все вхождения подстроки  $S_1$  на  $R_1$ ,  $S_2$  на  $R_2$  и т. д. Это простейший случай так называемого контекстного редактирования файла. Для представления строк исходного файла и пар строк  $(S_i, R_i)$  удобно применить АТД STRING, реализованный пакетом, приведенным в п. 5.2. Более того, там же мы рассмотрели процедуру REPLACE, выполняющую замену всех вхождений заданной подстроки в исходную строку. Будем предполагать, что эта процедура помещена в нашу личную библиотеку пакетов. Для представления пар  $(S_i, R_i)$  можно было бы применить массив строк переменной длины, но это потребовало бы зафиксировать максимальное количество таких пар при объявлении массива. Будем считать, что количество пар заранее не известно, оно может быть как весьма малым, так и довольно большим. Поэтому здесь целесообразно применить представление множества пар в виде списка, значениями элементов которого будут записи вида

RECORD S,R:STRING END

Для работы с таким списком, естественно, используем пакет обработки списков LIST, описанный выше. Исходные значения строк  $S_i, R_i$  должны вводиться из стандартного текстового файла INPUT. Поскольку это строки переменной длины, для их представления удобно выбрать такую форму, чтобы они могли вводиться процедурой STRINPUT из пакета STRINGIO. Значения вводимых строк

должны быть представлены в виде последовательностей символов, заключенных в апострофы, и должны следовать в таком порядке:  $S_1, R_1, S_2, R_2, \dots$ . Концом ввода будем считать пустую строку  $S_i$ . Заметим, что строки  $R_i$  могут быть пустыми строками. Замена на пустую строку означает просто удаление соответствующей подстроки  $S_i$ . Например, если редактируемый файл представляет собой текст на русском языке, содержимое файла INPUT могло бы иметь

```

PACKAGE LISTINPUT( L:INOUT );
PROCEDURE LISTINPUT( L:LIST );
  VAR P:VALTYPE;
  BEGIN
    L := NEWLIST;
    LOOP
      STRINPUT( INPUT,P.S );
      IF LENGTH(P.S) = 0 THEN EXIT;
      STRINPUT( INPUT,P.R );
      IF POS( P.S,P.R ) > 0 THEN
        WRITELN( P.S,' ЕСТЬ ПОДСТРОКА ',P.R)
      ELSE INCLUDE( P,L )
      END
    END;
  END;

```

Рис. 12. Процедура ввода списка замен

такой вид (стрелки введены только для удобства восприятия текста человеком, они будут игнорироваться процедурой STRINPUT):

```

'РИС. 5' ← 'РИС. 6'
'ПРИМ.' ← 'ПРИМЕЧАНИЕ'
'[32]' ← '[34]'
'[16]' ← "
"

```

На рис. 12 приведен текст процедуры LISTINPUT, предназначенный для ввода списка пар строк. Предполагается, что она погружена в контекст блока, к которому присоединены контексты пакетов обработки списков и строк. Проверка того, что  $S_i$  не является подстрокой  $R_i$ , необходима, чтобы предотвратить заикливание программы. Предположим, что эта процедура также занесена в личную библиотеку пакетов. На рис. 13 дан текст программы контекстного редактирования файла. Дадим к ней построчный комментарий.

- 1) Комментарий.
- 2) Заголовок программы.
- 3) Объявление текстовых файлов.

4) Присоединение пакета обработки строк с максимальной длиной строки, равной 120 символам, и пакета ввода-вывода строк.

5) Присоединение пакета обработки списков.

6) Объявление списка замен.

7) Объявление рабочей строки переменной длины для ввода в нее строки входного файла.

```
(* 1 *) -- ПРОГРАММА КОНТЕКСТНОГО РЕДАКТИРОВАНИЯ
(* 2 *) PROGRAM CONTEdit(INPUT,INFILE,OUTFILE);
(* 3 *)   VAR INFILE,OUTFILE:TEXT;
(* 4 *)   USE STRING( 120 ), STRINGIO( ,INFILE );
(* 5 *)   USE LIST( RECORD S,R:STRING END );
(* 6 *)   VAR Q:LIST;
(* 7 *)   ST:STRING;
(* 8 *)   E:ELEMENT;
(* 9 *)   USE REPLACE, LISTINPUT( Q );
(* 10 *)  BEGIN
(* 11 *)    LISTINPUT;
(* 12 *)    RESET(INFILE); REWRITE(OUTFILE);
(* 13 *)    WHILE NOT EOF(INFILE) DO
(* 14 *)      BEGIN
(* 15 *)        STRINPLINE(ST);
(* 16 *)        E := FIRSTEL(Q);
(* 17 *)        WHILE E <> NIL DO
(* 18 *)          BEGIN
(* 19 *)            REPLACE(ST,VALUE(E).S,VALUE(E).R);
(* 20 *)            E := NEXTEL(E,Q);
(* 21 *)          END;
(* 22 *)          STROUTPUT(OUTFILE,ST);
(* 23 *)          WRITELN(OUTFILE)
(* 24 *)        END
(* 25 *)      END.
```

Рис. 13. Пример АД-программы

8) Объявление рабочей переменной типа *элемент списка* для организации просмотра списка замен.

9) Присоединение текстов вспомогательных процедур.

10) Начало тела блока.

11) Ввод списка замен.

12) Открытие файлов.

13—14) Начало цикла обработки файла.

15) Чтение очередной строки.

- 16—18) Начало цикла просмотра списка замен.
- 19) Вызов процедуры REPLACE.
- 20) Переход к следующему элементу списка замен.
- 21) Конец цикла просмотра списка замен.
- 22—23) Вывод результирующей строки в выходной файл.
- 24) Завершение цикла обработки файла.
- 25) Конец программы.

Приведенная программа контекстного редактирования файла является хотя и весьма простой, но в полном смысле слова АТД-программой. Легко видеть, что ни в ней, ни в ее процедурах не используется ни одной переменной предопределенных в языке Паскаль типов. Исключение составляют только текстовые файлы, которые, впрочем, вполне обоснованно могут рассматриваться как объекты абстрактного типа TEXT. Разумеется, однако, что АТД-программирование ни в коей мере не запрещает использовать предопределенные типы. В данном случае они просто не понадобились.

## 5.6. Полиморфная процедура быстрой сортировки

Вернемся от АТД-пакетов к полиморфным процедурам. Напомним еще раз основное отличие их друг от друга. Если АТД применяются для представления абстрактных данных, то полиморфные процедуры — для абстрактного представления алгоритмов. Процесс преобразования обычной процедуры в полиморфную рассмотрим на примере известного алгоритма быстрой сортировки. Текст его приведен на рис. 14. Основная идея алгоритма в том, что в сортируемом массиве выбирается некоторый элемент (в данном варианте — это значение *среднего* элемента). Затем элементы массива попарно меняются местами так, что в одной части массива скапливаются элементы, меньшие среднего, а в другой — большие. Затем процедура рекурсивно вызывает сама себя два раза — для сортировки одной части массива и другой. После этого исходный массив отсортирован по возрастанию значений элементов. Разумеется, не каждую процедуру мы можем сделать полиморфной. Однако анализ текста процедуры быстрой сортировки на рис. 14 показывает, что в данном случае это возможно. В процедуре сделано только два предположения относительно типов обрабатываемых данных. Тип индекса предполагается целым. Тип элементов также целый. С помощью деструкторов типов вида INDEX(A) мы можем объявить индексные переменные так, что процедура сможет обрабатывать массивы с любыми типами индексов. Аналогично, и рабочие переменные X и W можно объявить с помощью деструктора ELEM(A). С такими изменениями процедура уже сможет обрабатывать массивы с любыми типами индексов и разными типами элементов (целый, вещественный, символьный, строковый, скалярный).

Можно расширить действие процедуры на массивы с любыми типами элементов, предусмотрев в ней вызов функции преобразования значения элемента в сравнимый ключ, аналогично тому, как это делалось в полиморфной процедуре поиска максимального элемента массива в п. 2.5. Текст процедуры быстрой сортировки, оформленный в виде пакета-функции, приведен на рис. 15. Пакет имеет два параметра-имени: имя сортируемого массива и имя функции преобразования значения элемента в ключ. В исходном варианте процедуры имя сортируемого массива не передается процедуре через

```

PROCEDURE QUICKSORT( L,H: INTEGER );
  VAR I,J: INTEGER;
      X,W: REAL;
  BEGIN
    I := L; J := H;
    X := A[ (I+J) DIV 2 ] ;
    REPEAT
      WHILE A[I] < X DO I := I+1;
      WHILE X < A[J] DO J := J-1;
      IF I <= J THEN
        BEGIN
          W := A[I]; A[I] := A[J]; A[J] := W;
          I := I+1;
          J := J-1
        END
      UNTIL I > J;
      IF L < J THEN QUICKSORT(L,J)
      IF I < H THEN QUICKSORT(I,H)
    END;
  
```

Рис. 14. Процедура быстрой сортировки

параметры. Вместо этого используется зафиксированное имя массива, объявленного в объемлющем блоке. Это сделано, видимо, из соображений эффективности, чтобы не передавать ссылку на один и тот же массив через параметры при многократных рекурсивных вызовах процедуры. Заметим, что это обстоятельство, крайне неудобное при традиционном подходе, требующее модификации текста процедуры (ручной или с помощью какого-либо макропроцессора или текстового редактора), всякий раз, как она должна использоваться для сортировки массива с другим именем, нас несколько не затрудняет. Настройка процедуры на нужное имя обрабатываемого массива будет производиться при присоединении пакета. Никакой модификации текста процедуры при этом не потребуются.



Обратим внимание также на некоторые изменения в выполнении операций над индексными переменными, вызванные тем, что мы не можем более полагать их целочисленными. Для увеличения или уменьшения индексных переменных на единицу используются встроенные функции языка Паскаль SUCC и PRED. Безусловно:

```
-- ПАКЕТ - ФУНКЦИЯ БЫСТРОЙ СОРТИРОВКИ
-- ВХОДНЫЕ ПАРАМЕТРЫ:
--   А - ИМЯ СОРТИРУЕМОГО МАССИВА (ЭЛЕМЕНТЫ И ИНДЕКСЫ
--   МОГУТ БЫТЬ ЛЮБОГО ТИПА)
--   KEY - ИМЯ ФУНКЦИИ, ПРЕОБРАЗУЮЩЕЙ ЭЛЕМЕНТ МАССИВА
--   В СРАВНИМЫЙ КЛЮЧ (ТИПА INTEGER, REAL, CHAR, SET)
```

```
PACKAGE QUICKSORT( A,KEY:INOUT );
PROCEDURE QUICKSORT( L,H:INDEX(A) );
  TYPE IND = INDEX(A);
  VAR I,J:IND;
      X :TYP(KEY);
      W :ELEM(A);
BEGIN
  I :=L; J := H;
  X := KEY( A[ IND((ORD(I)+ORD(J)) DIV 2) ] );
  REPEAT
    WHILE KEY( A[I] ) < X DO I := SUCC(I);
    WHILE X < KEY( A[J] ) DO J := PRED(J);
    IF I <= J THEN
      BEGIN
        W := A[I]; A[I] := A[J]; A[J] := W;
        IF I < LAST(I) THEN I := SUCC(I);
        IF J > FIRST(J) THEN J := PRED(J);
      END
    UNTIL I > J;
    IF L < J THEN QUICKSORT(L,J);
    IF I < H THEN QUICKSORT(I,H)
  END;
```

Рис. 15. Пакет-функция быстрой сортировки

уменьшение переменной J и увеличение переменной I после перестановки элементов заменено на условное, чтобы предотвратить случаи выхода их значений за границы, предписанные их типом (т. е. за границы массива). Хотя в данном случае этот выход за границы массива является вполне безопасным, так как эти значения не ис-

пользуются для индексации, но в языке Паскаль некорректным является уже само присваивание переменной значения, выходящего за границы, определенные ее типом. Оператор вычисления значения *среднего* элемента массива также модифицирован, так как индексные переменные произвольных типов не допускают выполнения над ними операций сложения и деления. К счастью, с помощью встроенной функции языка Паскаль ORD мы можем преобразовать значение переменной любого типа в целочисленное значение, произвести вычисления, а затем, используя имя типа как функцию преобразования типа, преобразовать полученное целое значение в значение нужного типа. Для тех, кого все же волнует эффективность, заметим, что время выполнения оператора от такой модификации несколько не увеличится, так как все эти преобразования делаются во время компиляции.

Рассмотрим несколько примеров использования этого пакета для сортировки массивов различных типов, в том числе и абстрактных.

1. Сортировка массива вещественных чисел.

```
VAR A:ARRAY[1..M] OF REAL;
FUNCTION PASS(X:REAL):REAL;
  BEGIN PASS := X END;
USE SORTA = QUICKSORT(A,PASS);

  . . .
SORTA(1,M);
```

2. Сортировка массива комплексных чисел по увеличению абсолютного значения элементов.

```
USE COMPLEX;
VAR B:ARRAY[M..N] OF COMPLEX;
USE QUICKSORT(B,CABS);

  . . .
QUICKSORT(M,N);
```

3. Сортировка массива комплексных чисел по увеличению мнимой части элементов производится так же, только раздел присоединения пакета имеет вид

```
USE QUICKSORT(B,IM);
```

4. Сортировка массива записей по увеличению суммы полей с именами ОКЛАД и ПРЕМИЯ.

```
VAR C:ARRAY[1..N] OF
  RECORD
```

```

      . . .
      ОКЛАД,ПРЕМИЯ:0..1000;
      . . .
      END;
      FUNCTION ЗАР(S:ELEM(C)):0..2000;
      BEGIN
        WITH S DO ЗАР := ОКЛАД+ПРЕМИЯ
      END;
      USE SORTC=QUICKSORT(C,ЗАР);
      . . .
      SORTC(1,N);

```

5. Сортировка массива строк переменной длины по увеличению текущей длины и по увеличению их строкового значения.

```

      USE STRING(80);
      VAR D:ARRAY[T] OF STRING;
      USE SORTDLEN=QUICKSORT(D,LENGTH),
        SORTDVAL=QUICKSORT(D,STRFIX);
      . . .
      SORTDLEN(FIRST(T).LAST(T));
      . . .
      SORTDVAL(FIRST(INDEX(D)),LAST(INDEX(D)));

```

Здесь SORTDLEN — имя процедуры сортировки массива D по длине элементов, SORTDVAL — имя процедуры сортировки по значению элементов (сравнение строк производится по правилам сравнения строковых значений языка Паскаль). Разный вид параметров вызова применен в чисто иллюстративных целях.

6. Сортировка строк матрицы по увеличению сумм элементов строк. Напомним, что двумерный массив в языке Паскаль трактуется как одномерный массив, элементами которого являются тоже одномерные массивы. В примере используется пакет VECTSUM, описанный в главе 2.

```

      VAR E:ARRAY[1..M,1..N] OF REAL;
      USE SUM=VECTSUM(ELEM(E));
      USE SORTE=QUICKSORT(E,SUM);
      . . .
      SORTE(1,M);

```

7. Сортировка строк матрицы по увеличению значения максимального элемента каждой строки.

```

      VAR F:ARRAY[1..M,1..N] OF INTEGER;
      USE MAXEL=MAXVALUE(ELEM(F),PASS);

```

```
USE SORTF=QUICKSORT(F,MAXEL);
```

```
..  
SORTF(I,M)
```

8. Сортировка строк матрицы множеств. В качестве ключа сортировки берется кардинальное число (количество элементов множества) объединения всех элементов строки. Для объединения элементов строки используется та же функция VECTSUM, что и в примере 6.

```
VAR G:ARRAY[T1,T2] OF SET OF T3;  
FUNCTION CARDINAL(S:ELEM(G)):INTEGER;  
    USE ROWSUM=VECTSUM(TYP(S));  
    BEGIN  
        CARDINAL := CARD(ROWSUM(S));  
    END;  
USE SORTG=QUICKSORT(G,CARDINAL);  
..  
SORTG(FIRST(T1),LAST(T1));
```

Примечания к примерам. В примерах 6 и 8 используется пакет VECTSUM, содержащий полиморфную функцию суммирования элементов вектора. Пакет имеет следующий интерфейс:

```
PACKAGE VECTSUM(VECTOR:TYPE);  
FUNCTION VECTSUM(A:VECTOR):ELEM(VECTOR);
```

Текст функции приведен в п. 2.5.

В примере 7 используется пакет MAXVALUE, содержащий полиморфную функцию поиска максимального значения элементов массива, почти аналогичную функции MAXINDEX из п. 2.5. Различие в том, что данная функция возвращает не индекс максимального элемента массива, а непосредственно его значение. Интерфейс пакета таков:

```
PACKAGE MAXVALUE(VECTOR:TYPE;KEY:INOUT);  
FUNCTION MAXVALUE(A:VECTOR):ELEM(VECTOR);
```

При его присоединении используется функция PASS из примера 1.

Столь большой набор примеров приведен здесь только лишь для того, чтобы продемонстрировать универсальность разработанной полиморфной процедуры. Без какого бы то ни было изменения исходного текста она применима к таким разнообразным случаям. Однако, как всегда, за универсальность мы платим эффективностью. В примере 1, например, использование функции преобразования элемента массива в сравнимый ключ излишне, так как вещественные элементы могли бы сравниваться непосредственно.

Сама же функция преобразования тривиальна, однако на ее многократные вызовы будет затрачиваться ощутимое время. В примерах 2—5 функции преобразования достаточно просты, так что их вызовы не приведут к значительным потерям времени. Однако они не совсем тривиальны, так что совсем без них обойтись нельзя. Примеры 6, 7 и 8 демонстрируют такую интересную возможность, как использование имени полиморфной процедуры, полученной путем настройки пакета на нужный тип, как входного параметра другого пакета. Однако эффективность сортировки здесь явно страдает, так как в примере 6, например, суммирование одной и той же строки матрицы будет производиться многократно.

При практической разработке полиморфных процедур, по-видимому, целесообразно применять подход, аналогичный взаимозаменяемости реализаций АТД. Разрабатывается одна универсальная процедура, применимая по возможности к самому широкому спектру типов, и несколько специализированных вариантов ее, более эффективных на ограниченном множестве типов. Если интерфейс всех вариантов пакета одинаков, что, конечно, достижимо далеко не всегда, то замена одной реализации процедуры на другую может производиться без изменения текста вызывающей программы. В случае процедуры сортировки, например, было бы целесообразно разработать четыре варианта пакета:

- 1) Универсальная полиморфная процедура.
- 2) Процедура, не использующая функцию преобразования элемента в ключ, применимая только к типам, допускающим операцию сравнения на неравенство.
- 3) Процедура с однократным вычислением ключа каждого элемента. Этот вариант может быть не менее универсален, чем первый, и взаимозаменяем с ним, но требует дополнительной памяти для хранения вычисленных ключей элементов массива.
- 4) Процедура сортировки массива записей по значению какого-либо поля. Сравниваемые поля должны иметь тип, допускающий операцию сравнения. Если это не так, то для сортировки массива записей может применяться универсальная процедура.

Второй вариант процедуры мы не рассматриваем из-за его тривиальности. Он получается из первого удалением вызова функции преобразования. Разработку третьего и четвертого вариантов можно предложить читателю в качестве упражнения.

Рассмотрим еще один пример, показывающий, как универсальная процедура сортировки может быть использована для построения ее специализированного варианта. Это полиморфная процедура сортировки списка. Она же служит еще одним примером использования средств пакета обработки списков. Процедура `LISTSORT`, определенная в пакете-функции `LISTSORT`, приведенном на рис. 16, предполагает, что списки строятся с помощью

пакета LIST, описанного выше. Параметрами пакета являются константа N, задающая максимально допустимый размер списка, и параметр-имя KEY, имеющий тот же смысл, что и ранее. Параметром полиморфной процедуры LISTSORT является имя сортируемого списка. В процедуре объявлен локальный массив ссылок,

```
-- ПАКЕТ-ФУНКЦИЯ СОРТИРОВКИ СПИСКА
-- ВХОДНЫЕ ПАРАМЕТРЫ ПАКЕТА:
-- N - МАКСИМАЛЬНАЯ ДЛИНА СОРТИРУЕМОГО СПИСКА
-- KEY - ФУНКЦИЯ ПРЕОБРАЗОВАНИЯ ЗНАЧЕНИЯ ЭЛЕМЕНТА
```

```
PACKAGE LISTSORT( N:CONST; KEY:INOUT );
PROCEDURE LISTSORT( L:LIST );
    VAR E:ELEMENT;
        V:VALTYPE;
        R:ARRAY[1..N] OF ELEMENT;
        J,K:INDEX(R);
    USE QUICKSORT( R,KEY );
    BEGIN
        E := FIRSTEL(L);
        J := 0;
        WHILE E <> NIL DO
            BEGIN
                J := J+1;
                IF J > N THEN . . . -- ОШИБКА
                R[J]:=E;
                V := REMOVED(E);
                E := FIRSTEL(L)
            END;
        IF J > 1 THEN QUICKSORT(1,J);
        FOR K := 1 TO J DO INCLUDE( R[K],L )
    END;
```

Рис. 16. Пакет сортировки списка

элементам которого присваиваются ссылки на все элементы списка в порядке их следования в списке. Заметим, что нулевой элемент массива всегда ссылается на заголовок списка, что необходимо для удобства переупорядочения списка после сортировки. Сортировка массива производится универсальной процедурой быстрой сортировки, а затем список переупорядочивается в соответствии с упорядоченностью элементов массива, т. е. элемент, на который

ссылается первый элемент массива, ставится на первое место, второй — на второе и т. д.

Рассмотрим пример использования пакета LISTSORT. Предположим, что значениями элементов списка являются целые числа. Тогда в качестве ключа сортировки может использоваться функция VALUE из пакета LIST, возвращающая значение элемента:

```
USE LIST(INTEGER),  
    LISTSORT(500,VALUE);  
VAR S:LIST;  
  
LISTSORT(S);
```

Заметим, что данная процедура является хотя и специализированной, но не менее полиморфной, чем универсальная. Она допускает произвольные типы элементов списка. Она использует только те средства пакета обработки списков, которые определены в его интерфейсе, т. е. она не зависит от конкретной реализации пакета LIST. Разумеется, знание деталей реализации списков могло бы сделать ее более эффективной. Возможно, такой вариант ее следовало бы включить в сам пакет обработки списков.

## 5.7. Пример построения программы методом пошаговой реорганизации

В качестве достаточно простого, но не слишком тривиального примера рассмотрим программу для решения головоломки «кубики-перевертыши», описанной в [15]. Суть игры описана так: «В простейшем варианте перевертыши — это восемь одинаковых кубиков, лежащих в квадратной коробочке. Каждый кубик окрашен в шесть цветов, по числу граней. Дно коробочки разделено на квадраты. В каждом квадрате (кроме одного) лежит по кубику. За счет свободной ячейки кубики можно последовательно перекачивать из квадрата в квадрат. Вынимать и переворачивать кубики не разрешается. В исходном состоянии цветные грани кубиков располагаются одинаково. В головоломке требуется изменить цвет верхних граней кубиков на любой другой цвет, например, с белого на синий. При этом цвета боковых граней кубика должны быть ориентированы одинаково» (рис. 17). Известно, что сложность игры не уступает знаменитому кубику Рубика. Существует также упрощенный вариант игры на поле размером  $2 \times 3$ . Наша цель — построить программу, которая бы находила кратчайшее решение этой головоломки. Эта задача выбрана, конечно, не из-за ее практической ценности, а из-за того, что на ее примере можно удачно продемонстрировать использование метода пошаговой реорганизации с использованием АТД, описанного выше.

Руководствуясь основными положениями метода пошаговой реорганизации, мы должны сначала выбрать наиболее удобные абстрактные типы данных и найти более или менее строгое решение в терминах операций над ними. При этом на первых шагах следует постараться забыть об эффективности программы. Попробуем сделать это. Единственный объект, которым мы манипулируем в процессе игры, есть игровое поле, характеризующееся своим состоянием,

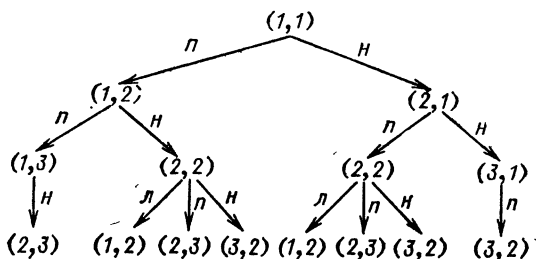
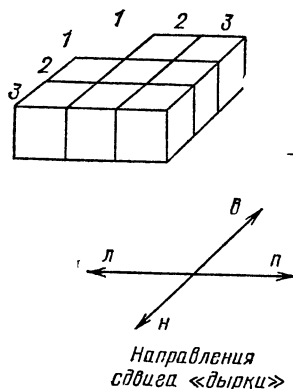


Рис. 17. Игровое поле и дерево решений трех первых ходов

т. е. ориентацией всех кубиков и положением свободного поля, которое будем для краткости называть *дыркой*. Основная операция над полем — перекатывание кубика в соседнюю с ним дырку. Эту операцию гораздо удобнее рассматривать не как перемещение кубика, а как сдвиг дырки по одному из возможных направлений: вправо, влево, вверх, вниз. При этом упрощается запись операций, так как не надо указывать, какой именно кубик перекатывается. Операция над состоянием однозначно определяется направлением сдвига дырки. Мы приходим к выводу, что для описания процесса поиска решения нам необходимы два типа: СОСТОЯНИЕ и НАПРАВЛЕНИЕ. В этой задаче представляется целесообразным использо-



вать русскую мнемонику вместо столь привычной большинству программистов английской. Тип НАПРАВЛЕНИЕ естественно представляется скалярным типом вида

ТИП НАПРАВЛЕНИЕ = (ВВЕРХ, ВНИЗ, ВПРАВО, ВЛЕВО);

Какие еще операции над этим типом нам потребуются, мы пока не знаем. Список операций над объектами типа СОСТОЯНИЕ представляется целесообразным сделать таким:

1) FUNCTION СДВИГ(S:СОСТОЯНИЕ;  
D:НАПРАВЛЕНИЕ):СОСТОЯНИЕ;

Сдвиг дырки в поле, находящемся в состоянии S, по направлению D;

2) FUNCTION НАЧАЛЬНОЕ:СОСТОЯНИЕ;  
Начальное состояние игрового поля;

3) FUNCTION КОНЕЧНОЕ(S:СОСТОЯНИЕ):BOOLEAN;  
Проверка того, не является ли данное состояние конечным;

4) FUNCTION ДОПУСТИМЫЕ(S:СОСТОЯНИЕ)  
:SET OF НАПРАВЛЕНИЕ;

Генерация множества направлений сдвига, допустимых из данного состояния.

Заметим сразу, что сдвиг из некоторого состояния «назад», т. е. в то состояние, откуда мы пришли в данное, следует считать недопустимым. Поэтому можно сразу зафиксировать некоторое свойство объектов типа СОСТОЯНИЕ. Они должны содержать информацию о том, путем какого сдвига мы в них пришли. Чтобы определить, что некоторое направление сдвига является недопустимым, надо уметь определять направление, обратное по отношению к данному. Поэтому к определению типа НАПРАВЛЕНИЕ надо добавить функцию

FUNCTION ОБРАТНОЕ(D:НАПРАВЛЕНИЕ):НАПРАВЛЕНИЕ;

/ Далее, разработка программы может идти по двум направлениям. Первое — разработка реализации АТД СОСТОЯНИЕ. Второе — попытка разработки общего алгоритма решения задачи и формулировка его в терминах определенных АТД СОСТОЯНИЕ и НАПРАВЛЕНИЕ. Возможно, при этом потребуются специфицировать и другие АТД. Выберем второй вариант, так как, прежде чем опускаться до реализации конкретных типов, надо попытаться определить все нужные типы и решить задачу в терминах операций над ними.

Основная операция над игровым полем, которая переводит его из одного состояния в другое, — операция сдвига дырки. Из некоторого состояния в зависимости от положения дырки допусти-

мы от одного (дырка в угловых позициях) до трех (дырка в центре) возможных сдвигов. Выбор направления очередного сдвига и есть тот момент, когда игрок принимает решение, влияющее на весь дальнейший ход игры. На рис. 17 показано так называемое дерево решений для нескольких первых ходов (для размера поля  $3 \times 3$ ). Здесь состояния обозначаются только положением дырки, задаваемой парой координат  $(I, J)$ . Направления сдвигов обозначены на стрелках буквами ( $v$  — вверх,  $n$  — вниз,  $p$  — вправо,  $l$  — влево). Будем помнить о том, что состояния, идентифицируемые на рисунке одинаковыми координатами дырки, на самом деле, конечно, неидентичны, так как в понятие состояния входит еще и ориентация всех кубиков. В задаче требуется найти кратчайший путь от корневой вершины к одной из вершин, которая удовлетворяет заданным свойствам конечной вершины. Поскольку метод пошаговой реорганизации рекомендует на каждом шаге выбирать простейший алгоритм, мы воспользуемся методом полного перебора. Он может быть использован здесь в двух модификациях. Первая — это *поиск вширь*, когда мы полностью проверяем все вершины  $N$ -го уровня дерева решений, прежде чем перейти к  $(N+1)$ -му. Применение этого варианта гарантирует нахождение кратчайшего пути, но требует много памяти, так как на каждом шаге необходимо хранить по крайней мере все вершины  $N$ -го и  $(N+1)$ -го уровней, количество которых при увеличении  $N$  растет в степенной зависимости. Если же мы хотим по достижении конечного состояния напечатать путь, который в него привел, надо либо хранить в памяти все пройденные уровни, либо в каждом состоянии хранить всю информацию о пути, который привел в него. Простейшие прикидки показывают, что для решения задачи размерностью  $3 \times 3$  этот метод, видимо, не приведет к решению из-за ограниченных возможностей современных машин. Второй вариант метода полного перебора — *поиск вглубь*. При этом мы спускаемся вниз по одному из возможных путей в дереве решений. Максимальная глубина спуска должна быть ограничена некоторой величиной  $L$ , подбираемой эмпирически в зависимости от некоторых априорных представлений о длине кратчайшего пути. При достижении этой максимальной глубины мы возвращаемся вверх по дереву до ближайшего разветвления с еще не опробованной ветвью и пытаемся спуститься по ней. В результате либо будет найдено некоторое решение с длиной пути, заведомо меньшей  $L$  (тогда можно попытаться уменьшить  $L$  до длины найденного пути минус 1 и найти более короткое решение), либо ни одного решения найдено не будет (тогда следует увеличить  $L$  и повторить поиск). Помня о том, что наша цель состоит не в том, чтобы решить задачу, а в том, чтобы продемонстрировать методику программирования, мы рассмотрим оба эти варианта.

Начнем с первого. На первом его шаге берется начальное состояние и генерируются все состояния, достижимые из него ровно за один сдвиг дырки. Затем, просматривая это множество состояний, мы создаем все состояния, достижимые еще за один сдвиг и т. д. В общем виде, используя множество состояний, достижимых ровно за  $n$  шагов (назовем это множество ТЕКУЩИЕ), мы создаем новое множество состояний, достижимых за  $n+1$  шаг (назовем его НОВЫЕ). После создания каждого нового состояния, естественно, надо проверить, не конечное ли оно, и, если это так, перейти на печать найденного пути. Заметим, что формулировку алгоритма мы даем, используя термин *множество состояний*. Поэтому мы приходим к выводу, что нам требуется определить еще один АТД — МНОЖЕСТВО. Операции над объектами этого типа определим таким образом:

1) FUNCTION ПУСТО:МНОЖЕСТВО;

Создание нового пустого множества;

2) PROCEDURE ВКЛЮЧИТЬ(S:СОСТОЯНИЕ;

VAR A:МНОЖЕСТВО);

Включение в множество нового состояния;

3) PROCEDURE ПРИСВОИТЬ(VAR A,B:МНОЖЕСТВО);

Операция присваивания множества B множеству A;

4) PROCEDURE ДЛЯ ВЕХ(A:МНОЖЕСТВО;

PROCEDURE P(S:СОСТОЯНИЕ));

Выполнение процедуры P над всеми элементами множества.

Необходимость последней процедуры следует оговорить особо. Дело в том, что без знания деталей реализации АТД МНОЖЕСТВО невозможно построить перебор всех его элементов. Поэтому и необходима специальная процедура — итератор, которая, будучи размещенной в теле пакета, определяющего АТД, имеет доступ к информации о его реализации.

На рис. 18 приведен текст *нулевого приближения*. Программа использует три АТД-пакета (НАПРАВЛЕНИЕ, СОСТОЯНИЕ и МНОЖЕСТВО). Предполагается, что пакет СОСТОЯНИЕ, кроме определенных ранее операций, включает в себя и процедуру печати пути, приведшего к данному состоянию. Внутренняя процедура СДВИГИ выполняет все сдвиги, допустимые из данного состояния, и включает новые сгенерированные состояния в множество НОВЫЕ. Каждое новое состояние проверяется с помощью функции КОНЕЧНОЕ из пакета СОСТОЯНИЕ. Если найдено искомое состояние, то выполняется выход на процедуру печати, причем найденное состояние сохраняется в качестве значения переменной СЛЕДУЮЩЕЕ.

Заметим, что эта программа является вполне завершенной программой на языке АТ-Паскаль, хотя мы и не определили еще реализацию АТД. Ее можно даже компилировать, чтобы убедиться в корректности использования операций над АТД. Для этого надо в АТД-пакетах объявить типы произвольным образом (например,

```

PROGRAM КУБИК1(OUTPUT);
LABEL 1;
USE НАПРАВЛЕНИЕ, СОСТОЯНИЕ, МНОЖЕСТВО;
VAR ТЕКУЩИЕ, НОВЫЕ: МНОЖЕСТВО;
    СЛЕДУЮЩЕЕ: СОСТОЯНИЕ;
PROCEDURE СДВИГИ(S: СОСТОЯНИЕ);
    VAR НАПР: НАПРАВЛЕНИЕ;
BEGIN
    FORALL НАПР IN ДОПУСТИМЫЕ(S) DO
    BEGIN
        СЛЕДУЮЩЕЕ := СДВИГ(НАПР);
        IF КОНЕЧНОЕ(СЛЕДУЮЩЕЕ) THEN GOTO 1
        ELSE ВКЛЮЧИТЬ(СЛЕДУЮЩЕЕ, НОВЫЕ)
    END
END;
BEGIN
    ТЕКУЩИЕ := ПУСТО;
    ВКЛЮЧИТЬ(ИСХОДНОЕ, ТЕКУЩИЕ);
    НОВЫЕ := ПУСТО;
    LOOP
        ДЛЯ ВСЕХ(ТЕКУЩИЕ, СДВИГИ);
        ПРИСВОИТЬ(ТЕКУЩИЕ, НОВЫЕ);
        НОВЫЕ := ПУСТО;
    END;
    1: ПЕЧАТЬ(СЛЕДУЮЩЕЕ)
END.

```

Рис. 18. Первый вариант программы (с поиском «вширь»)

как INTEGER) и задать интерфейсы процедур, объявив их с пустыми телами или как внешние процедуры. Заметим также, что эта программа является в полном смысле слова АТД-программой. В ней используются исключительно операции над АТД. Примечательно, что при ее составлении нам не понадобилось ни одной операции над предопределенными в языке Паскаль типами.

Рассмотрим теперь реализацию использованных в программе АТД. Тип НАПРАВЛЕНИЕ совершенно естественно и просто реа-

лизуется скалярным типом. Текст пакета НАПРАВЛЕНИЕ показан на рис. 19. Для реализации типа МНОЖЕСТВО на данном этапе разработки программы естественно воспользоваться списковым представлением множеств. Поскольку у нас уже был разработан параметризуемый пакет LIST для операций над списками, им мы и воспользуемся. Текст пакета показан на рис. 20. Разумеется, было бы эффективнее вместо типа МНОЖЕСТВО использовать непосредственно тип LIST или в процедурах пакета МНОЖЕСТВО для работы со списковыми структурами использовать не АТД, а явные

**PACKAGE НАПРАВЛЕНИЕ;**

**TYPE НАПРАВЛЕНИЕ=(ВПРАВО,ВЛЕВО,ВВЕРХ,ВНИЗ);**

**FUNCTION ОБРАТНОЕ(D:НАПРАВЛЕНИЕ):НАПРАВЛЕНИЕ;**

**BEGIN**

**CASE D OF**

**ВПРАВО:ОБРАТНОЕ:=ВЛЕВО;**

**ВЛЕВО: ОБРАТНОЕ:=ВПРАВО;**

**ВВЕРХ: ОБРАТНОЕ:=ВНИЗ;**

**ВНИЗ: ОБРАТНОЕ:=ВВЕРХ;**

**END**

**END;**

Рис. 19. Реализация АТД НАПРАВЛЕНИЕ

операции над указателями, но на данном этапе мы стремимся не к эффективности, а к простоте. Эти же две возможности отметим на будущее на случай возможного совершенствования программы.

Реализация АТД СОСТОЯНИЕ более сложна, чем предыдущих. Как мы уже отмечали, состояние игрового поля полностью характеризуется положением дырки и ориентацией всех кубиков. Дополнительно в объект типа СОСТОЯНИЕ должна быть включена ссылка на то состояние, из которого мы пришли в данное, и направление сдвига, посредством которого это было сделано. Это необходимо, чтобы исключить из числа допустимых сдвиг «назад» и чтобы можно было при нахождении решения вывести всю цепочку сдвигов от начального состояния к конечному. Таким образом, все узлы дерева решений, пример которого показан на рис. 17, оказываются связанными ссылками, идущими от *листьев* к *корню*. Как видим, наше представление дерева отличается от общепринятого, где ссылки идут от *корня* к *листьям*. Объявление типа СОСТОЯНИЕ будет выглядеть так:

**TYPE СОСТОЯНИЕ=@СОСТ;**

**СОСТ=RECORD**

**ПРЕДЫДУЩЕЕ:СОСТОЯНИЕ;**

**НАПР :НАПРАВЛЕНИЕ;**

ДЫРКА	:RECORD
	I:1..N;
	J:1..M
	END;
ПОЛЕ	:ARRAY[1..N,1..M] OF КУБИК
END:	

Здесь константы N и M — размеры игрового поля. Обычно N=2 или N=3, а M=3, хотя программу следует строить так, чтобы была

PACKAGE МНОЖЕСТВО;

USE МНОЖЕСТВО=LIST(СОСТОЯНИЕ);

FUNCTION ПУСТО:МНОЖЕСТВО;

BEGIN ПУСТО:=МНОЖЕСТВО.NEWLIST END;

PROCEDURE ПРИСВОИТЬ(VAR A,B:МНОЖЕСТВО);

BEGIN

МНОЖЕСТВО.DESTROY(A);

A:=B

END;

PROCEDURE ВКЛЮЧИТЬ(S:СОСТОЯНИЕ; A:МНОЖЕСТВО);

BEGIN

МНОЖЕСТВО.INCLUDE(S,A)

END;

PROCEDURE ДЛЯВСЕХ(A:МНОЖЕСТВО; PROCEDURE P(S:СОСТОЯНИЕ));

VAR E:МНОЖЕСТВО.ELEMENT;

BEGIN

E:=МНОЖЕСТВО.FIRSTEL(A);

WHILE E <> NIL DO

BEGIN

P(МНОЖЕСТВО.VALUE(E));

E:=МНОЖЕСТВО.NEXTEL(E,A)

END

END;

Рис. 20. Реализация АТД МНОЖЕСТВО

возможность рассматривать и задачи больших размерностей. Важной особенностью такого определения типа СОСТОЯНИЕ является то, что значениями переменных этого типа являются не сами состояния, а ссылки на записи, характеризующие состояния. Хотя в пакете МНОЖЕСТВО множество было определено как список состояний, на самом деле — это список ссылок. Это важно, поскольку при выполнении операции ПРИСВОИТЬ над списком (см. рис. 20) процедура DESTROY будет уничтожать не сами

объекты-состояния, а только ссылки на них. Все сгенерированные в процессе поиска состояния будут оставаться в памяти до конца работы программы, что и необходимо для печати найденного пути.

При определении типа СОСТОЯНИЕ мы ввели и использовали новый тип КУБИК. Объект этого типа представляет собой один кубик на игровом поле. Представляется целесообразным отделить операции над кубиком от операций над всем полем, так как это упрощает определение типа СОСТОЯНИЕ и позволяет в дальнейшем оптимизировать реализацию типов СОСТОЯНИЕ и КУБИК независимо друг от друга. Прежде чем заняться реализацией типа СОСТОЯНИЕ, надо зафиксировать операции над типом КУБИК.

1) CONST НАЧПОЛОЖЕНИЕ

Константа, задающая начальное положение всех кубиков.

2) FUNCTION ОРИЕНТАЦИЯ(C1, C2:КУБИК):BOOLEAN;

Операция проверки совпадения ориентаций двух кубиков. Значение функции равно TRUE, если кубики ориентированы одинаково.

3) FUNCTION ПОВОРОТ(C:КУБИК;

D:НАПРАВЛЕНИЕ):КУБИК;

Операция перекачивания кубика в заданном направлении. Значением функции является кубик с новой ориентацией, получившейся после перекачивания.

Оставим реализацию типа КУБИК на потом, а сейчас займемся реализацией типа СОСТОЯНИЕ. Функция ИСХОДНОЕ реализуется тривиально. Заметим только, что здесь и далее ориентация позиции поля, координаты которой совпадают с текущими координатами дырки, не играет никакой роли, поэтому в этой позиции может находиться фиктивный кубик с произвольной ориентацией. Функция КОНЕЧНОЕ также тривиальна, она просто сравнивает ориентации всех кубиков с ориентацией одного из них и, если находит несовпадение, возвращает результат FALSE. Единственная сложность в ней состоит в том, что при анализе очередной позиции поля необходимо проверять, не дырка ли это. Эту сложность можно устранить, сделав так, чтобы ориентация фиктивного кубика, находящегося в позиции дырки, всегда совпадала с ориентацией последнего перемещенного кубика. Тогда, если на каком-то шаге мы пришли к конечному состоянию, ориентация кубиков во всех позициях, включая и позицию дырки, окажется одинаковой. Мы учтем это при определении операции СДВИГ. В ней на основе координат дырки и направления сдвига вычисляются координаты дырки после сдвига. Затем выполняется поворот кубика, имеющего эти координаты (так как именно на его месте должна оказаться дырка после сдвига), по направлению, обратному по отноше-

нию к направлению сдвига дырки. Полученный при повороте кубик помещается на место старой дырки. Кроме того, по указанным выше причинам мы помещаем его и в позицию новой дырки. На этом выполнение операции сдвига дырки завершается. Функция ДОПУСТИМЫЕ, генерирующая множество допустимых направлений сдвига от данного положения дырки, тоже достаточно проста. В ней из множества четырех первоначально допустимых направлений вычитаются те, которые запрещены положением дырки на краю поля или направлением предыдущего сдвига. Заметим, что простой модификацией одной только этой процедуры мы можем превратить квадратное игровое поле в цилиндрическое или даже в тороидальное. Хотя такие варианты игры вряд ли могут быть осуществлены на практике, на нашей абстрактной модели было бы вполне интересно рассмотреть и их. Реализация процедуры ПЕЧАТЬ также требует определения еще одного АТД для выполнения операций над стеком. Дело в том, что пройти по дереву решений мы можем только в одном направлении — снизу вверх (от *листьев* к *корню*), а печатать решение надо, естественно, в порядке движения от исходного состояния к конечному, т. е. от *корня* к *листу*. Конечно, в данном случае можно было бы обойтись простым массивом. Использование стека здесь диктуется чисто иллюстративными целями. Кроме того, АТД STACKVAR входит в число так называемых стандартных пакетов языка АТ-Паскаль. Пакет STACKVAR определяет не тип, а одну локальную стековую переменную (см. п. 4.4 и рис. 4) и операции над ней:

1) PROCEDURE INIT;

Инициализация стека;

2) PROCEDURE PUSH(E:T);

Занесение нового элемента типа T в вершину стека;

3) PROCEDURE POP(VAR E:T);

Выталкивание верхнего элемента стека;

4) FUNCTION DEPTH:INTEGER;

Глубина стека, т. е. количество элементов в нем;

5) FUNCTION TOP:T;

Верхний элемент стека.

Текст пакета СОСТОЯНИЕ показан на рис. 21. Полную реализацию пакета STACKVAR оставим в качестве упражнения для читателя.

Чтобы завершить построение программы, нам осталось реализовать АТД КУБИК. По условиям задачи грани кубика окрашены в шесть различных цветов. Возможна также трехцветная окраска, где противоположные грани окрашены одинаково. Поэтому введем



```

PACKAGE СОСТОЯНИЕ;
CONST N=3; M=3;
USE КУБИК;
PRIVATE TYPE СОСТОЯНИЕ=@СОСТ;
    СОСТ=RECORD
        ПРЕДЫДУЩЕЕ:СОСТОЯНИЕ;
        НАПР      :НАПРАВЛЕНИЕ;
        ДЫРКА     :RECORD
            I:1..N;
            J:1..M
        END;
        ПОЛЕ      :ARRAY[1..N,1..M] OF КУБИК
    END
FUNCTION ИСХОДНОЕ:СОСТОЯНИЕ;
VAR I:INDEX(СОСТ.ПОЛЕ);
    J:INDEX(СОСТ.ПОЛЕ[]);
    S:СОСТОЯНИЕ;
BEGIN
    NEW(S);
    WITH S@ DO
    BEGIN
        ПРЕДЫДУЩЕЕ:=NIL;
        НАПР:=ВНИЗ;
        ДЫРКА.I:=1; ДЫРКА.J:=1;
        FOR I,J DO ПОЛЕ[I,J]:=НАЧПОЛОЖЕНИЕ
    END;
    ИСХОДНОЕ:=S
END;

FUNCTION КОНЕЧНОЕ(S:СОСТОЯНИЕ):BOOLEAN;
LABEL 1;
VAR I:INDEX(СОСТ.ПОЛЕ);
    J:INDEX(СОСТ.ПОЛЕ[]);
    ПЕРВЫЙ:КУБИК;

```

Рис. 21. Реализация АД СОСТОЯНИЕ

```

BEGIN
  ПЕРВЫЙ:=S.ПОЛЕ[1,1]; КОНЕЧНОЕ:=FALSE;
  FOR I,J DO
    IF NOT ОРИЕНТАЦИЯ(S.ПОЛЕ[I,J],ПЕРВЫЙ) THEN GOTO 1;
    КОНЕЧНОЕ:=TRUE;
  1:END;
FUNCTION СДВИГ(S:СОСТОЯНИЕ; D:НАПРАВЛЕНИЕ):СОСТОЯНИЕ;
VAR Q:СОСТОЯНИЕ;
    C:КУБИК;
BEGIN
  NEW(Q); Q@:=S@;
  WITH Q@ DO
    BEGIN
      ПРЕДЫДУЩЕЕ:=S; НАПР:=D;
      WITH ДЫРКА DO
        BEGIN
          CASE D OF
            ВПРАВО: J:=SUCC(S.ДЫРКА.J);
            ВЛЕВО: J:=PRED(S.ДЫРКА.J);
            ВВЕРХ: I:=PRED(S.ДЫРКА.I);
            ВНИЗ: I:=SUCC(S.ДЫРКА.I)
          ..
        END;
        C:=ПОВОРОТ(S.ПОЛЕ[I,J],ОБРАТНОЕ(D));
        ПОЛЕ[S.ДЫРКА.I,S.ДЫРКА.J]:=C;
        ПОЛЕ[I,J]:=C
      END;
    END;
  СДВИГ:=Q
END;
PROCEDURE ПЕЧАТЬ(R:СОСТОЯНИЕ);
  USE S=STACKVAR(НАПРАВЛЕНИЕ,100);
  VAR H:НАПРАВЛЕНИЕ;

```

Рис. 21. Реализация АТД СОСТОЯНИЕ

```

BEGIN
  S.INIT;
  WHILE R <> NIL DO
  BEGIN
    S.PUSH(R.НАПР);
    R:=R.ПРЕДЫДУЩЕЕ
  END;
  S.POP(H);
  WHILE DEPTH(S) > 0 DO
  BEGIN
    S.POP(H);
    CASE H OF
      ВПРАВО: WRITE(' ВПРАВО ');
      ВЛЕВО:  WRITE(' ВЛЕВО  ');
      ВВЕРХ:  WRITE(' ВВЕРХ  ');
      ВНИЗ:   WRITE(' ВНИЗ   ');
    END
  END
END;

PROCEDURE ПЕЧАТЬСОСТОЯНИЯ(S:СОСТОЯНИЕ);
  VAR I:INDEX(СОСТ.ПОЛЕ);
      J:INDEX(СОСТ.ПОЛЕ[]);
BEGIN
  FOR I,J DO WRITE(ORD(S.ПОЛЕ[I,J] [ВЕРХНЯЯ]):3);
  Writeln
END;

```

Рис. 21. Реализация АТД СОСТОЯНИЕ

константу КВОЦВЕТОВ и скалярный тип ЦВЕТ. Определим тип КУБИК так:

```

TYPE КУБИК=PACKED
  ARRAY[(ПРАВАЯ,ПЕРЕДНЯЯ,ВЕРХНЯЯ)] OF ЦВЕТ;

```

Заметим, что ориентация кубика однозначно задается окраской трех «видимых» его граней (правой, передней и верхней).

Вычисление цвета противоположной грани осуществляется функцией ПРОТИВ. При КВОЦВЕТОВ=3 цвет противоположной грани равен цвету исходной. Реализация функции ОРИЕНТАЦИЯ основана на том факте, что тип КУБИК в памяти представлен

```

PACKAGE КУБИК;
CONST КВОЦВЕТОВ=8;
LOCAL TYPE ЦВЕТ=(КРАСН, БЕЛ, СИН, ЖЕЛТ, КОРИЧН, ЗЕЛЕН);
PRIVATE TYPE КУБИК=ARRAY[(ПРАВАЯ, ПЕРЕДНЯЯ, ВЕРХНЯЯ)]
                                OF ЦВЕТ;
CONST НАЧПОЛОЖЕНИЕ=(# КРАСН, БЕЛ, СИН #):КУБИК;
FUNCTION ОРИЕНТАЦИЯ(C1, C2:КУБИК):BOOLEAN;
    TYPE ORIENT=PACKED ARRAY[1..3] OF CHAR;
BEGIN
    ОРИЕНТАЦИЯ:=ORIENT(C1)=ORIENT(C2)
END;
FUNCTION ПРОТИВ(A:ЦВЕТ):ЦВЕТ;
BEGIN
    ПРОТИВ:=(ORD(A)+3) MOD КВОЦВЕТОВ
END;
FUNCTION ПОВОРОТ(C:КУБИК; D:НАПРАВЛЕНИЕ):КУБИК;
VAR A:КУБИК;
BEGIN
    CASE D OF
        ВПРАВО: BEGIN
            A[ПРАВАЯ]:=C[ВЕРХНЯЯ];
            A[ПЕРЕДНЯЯ]:=C[ПЕРЕДНЯЯ];
            A[ВЕРХНЯЯ]:=ПРОТИВ(C[ПРАВАЯ])
        END;
        ВЛЕВО: BEGIN
            A[ПРАВАЯ]:=ПРОТИВ(C[ВЕРХНЯЯ]);
            A[ПЕРЕДНЯЯ]:=C[ПЕРЕДНЯЯ];
            A[ВЕРХНЯЯ]:=C[ПРАВАЯ]
        END;
        ВВЕРХ: BEGIN
            A[ПРАВАЯ]:=C[ПРАВАЯ];
            A[ПЕРЕДНЯЯ]:=ПРОТИВ(C[ВЕРХНЯЯ]);
            A[ВЕРХНЯЯ]:=C[ПЕРЕДНЯЯ]
        END;
    END;
END;

```

Рис. 22. Реализация АДД КУБИК

```

ВНИЗ: BEGIN
    A[ПРАВАЯ]:=C[ПРАВАЯ];
    A[ПЕРЕДНЯЯ]:=C[ВЕРХНЯЯ];
    A[ВЕРХНЯЯ]:=ПРОТИВ(C[ПЕРЕДНЯЯ])
END;
END;
ПОВОРОТ:=A
END;

```

Рис. 22. Реализация АТД КУБИК

тремя последовательными байтами, что позволяет сравнивать кубики как символьные строки, используя процедуру преобразования типа. В функции ПОВОРОТ окраска правой, передней и верхней граней кубика, получаемая после поворота кубика, вычисляется, исходя из окраски граней исходного кубика и направления поворота. Например, при повороте вправо передняя грань остается на месте, верхняя грань перемещается на место правой, а левая грань становится верхней. Аналогично, воображая в уме кубик и его поворот через грань, мы записываем преобразования и для остальных трех направлений сдвига. Текст пакета КУБИК показан на рис. 22. Достоинством такого способа описания ориентации кубика является естественность и простота, хотя, конечно, возможны и более эффективные реализации. Например, поскольку имеется всего 24 ориентации шестичветного кубика, можно было бы задать операцию ПОВОРОТ в табличном виде в форме массива

```

TYPE ОРИЕНТАЦИЯ = 1..24;
VAR ПОВОРОТ:ARRAY[ОРИЕНТАЦИЯ, НАПРАВЛЕНИЕ]
    OF ОРИЕНТАЦИЯ;

```

Значения 96 элементов этого массива можно задать либо вручную, либо программно.

На этом разработка *нулевого* приближения программы завершается. Справедливости ради следует сказать, что реальный процесс ее разработки не всегда протекал так гладко, как это было описано. Что, впрочем, представляется совершенно естественным, так как процесс построения программы, использующей некоторые АТД, должен идти одновременно со спецификацией самих АТД. Именно в этом случае удается подобрать наиболее удачный набор операций. Вместе с тем тщательная проработка программы на этапе разработки позволила практически исключить этап отладки. После устранения нескольких опечаток, сделанных при вводе текста программы, она заработала практически сразу и было найдено решение для случая трехцветного кубика на поле размером  $2 \times 3$ . Длина найденного решения — 15 ходов. Вместе с тем было обнаружено, что попытки решения задачи для шестичветного кубика даже

на поле  $2 \times 3$  оканчиваются неудачей из-за нехватки оперативной памяти (при разумных ограничениях на объем памяти — 500 К). Включенная в программу отладочная печать показала, что этого объема хватает всего на 22—23 шага.

У нас в запасе остались два возможных пути решения задачи: поиск «вглубь» и хранение в каждом состоянии всего пути, чтобы не хранить в памяти все дерево решений. Рассмотрим первый из них. Легко видеть, что вариант программы с поиском «вглубь» может быть построен из тех же «строительных блоков», что и первый вариант, т. е. из АТД НАПРАВЛЕНИЕ, СОСТОЯНИЕ и КУБИК. Пакет МНОЖЕСТВО в ней не нужен. Текст этого варианта представлен на рис. 23. Вся логика программы сосредоточена во внутренней рекурсивной процедуре ПОИСК. Параметр S этой процедуры есть состояние, из которого начинается поиск решения. Параметр УРОВЕНЬ — уровень дерева решений, на котором находится это состояние. При каждом вызове процедуры УРОВЕНЬ увеличивается на единицу и сравнивается с заданной величиной ПРЕДЕЛ, которая ограничивает глубину просмотра дерева решений. С помощью стандартных процедур MARK и RELEASE обеспечивается освобождение памяти при возврате из процедуры, если какой-либо путь просмотрен до заданного предела и решения не найдено. Поскольку, как было сказано, этот алгоритм не гарантирует нахождение именно кратчайшего пути, целесообразно искать и печатать все решения, длина которых не превосходит заданного предела. Поскольку может оказаться, что в результате манипуляций над полем мы пришли опять к исходному состоянию, целесообразно ввести в пакет СОСТОЯНИЕ процедуру ПЕЧАТЬСОСТОЯНИЯ и печатать не только путь, приведший к конечному состоянию, но и само это состояние для сравнения его с исходным, которое также печатается в начале прогона. Хотелось бы надеяться, что у читателя не возникло здесь вопроса: «Зачем вводить новую процедуру, тело которой состоит всего из двух операторов? Не проще ли включить эти операторы в текст процедуры ПОИСК?» Да, конечно, но тогда главная программа стала бы зависимой от реализации АТД СОСТОЯНИЕ, чего следует избегать. Чтобы оградить самого себя и читателей от подобных соблазнов, тип СОСТ в пакете СОСТОЯНИЕ объявлен приватным. Поэтому включение оператора печати состояния в процедуру ПОИСК было бы воспринято компилятором как ошибка.

Какой бы то ни было отладки программы КУБИК2 также не потребовалось. На поле размером  $2 \times 3$  для трехцветного кубика было получено решение, найденное ранее первым вариантом программы. Для шестицветного кубика на поле  $2 \times 3$  было найдено множество решений длиной менее 50 ходов, кратчайшее из которых имеет длину 37 ходов. Среднее время процессора ЕС-1033 на поиск

одного решения составляет около 4 минут. Требуемый объем памяти — 70 К.

Заметим, что порядок просмотра дерева решений определяется порядком перечисления направлений в объявлении типа НАПРАВ-

```
PROGRAM КУБИК2(INPUT,OUTPUT);
LABEL 1;
USE НАПРАВЛЕНИЕ,СОСТОЯНИЕ;
VAR ПРЕДЕЛ: INTEGER;
    ИСКОМОЕ: СОСТОЯНИЕ;
PROCEDURE ПОИСК(УРОВЕНЬ: INTEGER; S: СОСТОЯНИЕ);
    VAR НАПР: НАПРАВЛЕНИЕ;
        НОВОЕ: СОСТОЯНИЕ;
        ПАМЯТЬ: REF( INTEGER);
BEGIN
    MARK(ПАМЯТЬ);
    IF УРОВЕНЬ <= ПРЕДЕЛ THEN
        FORALL НАПР IN ДОПУСТИМЫЕ(S) DO
            BEGIN
                НОВОЕ:=СДВИГ(S,НАПР);
                IF КОНЕЧНОЕ(НОВОЕ) THEN
                    BEGIN
                        ПЕЧАТЬ(НОВОЕ); ПЕЧАТЬСОСТОЯНИЯ(НОВОЕ)
                    END
                ELSE ПОИСК(УРОВЕНЬ+1,НОВОЕ)
            END;
        RELEASE(ПАМЯТЬ)
    END;
BEGIN
    READ(ПРЕДЕЛ);
    ПЕЧАТЬ(ИСХОДНОЕ);
    ПОИСК(0,ИСХОДНОЕ);
END.
```

Рис. 23. Второй вариант программы (с поиском «вглубь»)

ЛЕНИЕ. Поэтому был проведен ряд экспериментов при измененном порядке просмотра дерева. В частности, указанное выше решение длиной в 37 ходов было найдено при объявлении направлений в порядке (вверх, вниз, вправо, влево). Вместе с тем попытки решения задачи на поле размером  $3 \times 3$  не привели к успеху за разумное время порядка 1 часа. Решение этой задачи, как видно, требует дальнейшей оптимизации программы,

## ГЛАВА 6

### ВВЕДЕНИЕ

### В РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ

#### 6.1. Основные понятия распределенного программирования

До сих пор мы рассматривали средства и приемы АТД-программирования, оставаясь в рамках традиционного последовательного программирования. Теперь же мы вступаем в новую, мало изученную область — область распределенного программирования (*distributed programming*), возникшую как самостоятельное направление развития языков и методов программирования около десяти лет назад. Мы будем использовать следующее, возможно, не слишком строгое определение: *распределенное программирование* — это совокупность языковых средств и методов программирования систем распределенной обработки данных в сетях ЭВМ и многомашинных комплексах. Под системой распределенной обработки данных здесь понимается такая система, отдельные компоненты которой одновременно функционируют на разных ЭВМ, обладающих средствами обмена данными друг с другом. Одновременность функционирования разных компонент распределенной системы играет важную роль в этом определении. Без нее в разряд распределенных систем могла бы попасть и такая, например, система, где данные, полученные с помощью одной программы, записываются на магнитную ленту, которая перевозится на другой вычислительный центр и обрабатывается там другой программой. Такая система обработки данных не является, конечно, распределенной системой. Но вот, если каждый блок данных, сгенерированный первой программой, сразу же передается по линиям связи на другую ЭВМ, где обрабатывается другой выполняемой в то же время программой, это уже пусть и простейшая, но действительно распределенная система. Мы намеренно говорим здесь об одновременности, а не о параллельности выполнения компонент распределенной системы, так как термин «параллельность» имеет слишком широкое значение и часто



употребляется в разных смыслах. Имеет смысл более четко выделить положение распределенного программирования среди других направлений развития современного программирования.

Однако, прежде чем сделать это, необходимо сформулировать понятие процесса — основное понятие распределенного программирования. Программа — объект статический. Одна и та же программа может существовать в разных видах: в виде текста на бумаге, колоды перфокарт, набора данных на магнитном диске. При этом она остается одной программой. Копирование программы с одной магнитной ленты на другую не порождает новой программы, программа осталась той же, просто возникла еще одна точная ее копия. Процесс же, в отличие от программы, — объект динамический. Он возникает только тогда, когда программа начинает выполняться некоторым *процессором*, будь то ЭВМ или человек, проигрывающий работу программы в уме. Когда выполнение программы доходит до конца, процесс исчезает, остаются только его результаты. Можно одновременно запустить на выполнение на одной машине одну и ту же программу два раза, и это будут два разных процесса, хотя программа у них и одна.

Неотъемлемым свойством процесса является то, что в каждый момент своего существования процесс обладает состоянием, характеризующим тем местом программы, которое достигнуто к некоторому моменту времени, и теми значениями, которые получили переменные к этому моменту. Заметим, что у каждого процесса есть свой набор собственных переменных, хотя иногда некоторые переменные могут быть общими для разных процессов. Это похоже на то, как при каждом вызове процедуры в памяти отводится место под ее локальные переменные. Кстати, рекурсивные процедуры и являются примером таких языковых средств, при изучении которых каждый программист начинает ощущать, что программа и процесс — это далеко не одно и то же.

Вводить и использовать понятие процесса имеет смысл только тогда, когда имеется несколько процессов, которые могут взаимодействовать друг с другом для решения некоторой общей для них задачи. В соответствии с видами средств взаимодействия процессов, способами организации процессов и целями их создания можно выделить четыре основных направления развития современного программирования.

1. *Последовательное программирование* — это всем хорошо знакомое традиционное программирование. В нем на понятии процесса внимание, как правило, не акцентируется.

2. *Параллельное программирование* — совокупность языковых средств и методов решения задач на ЭВМ, допускающих параллельную обработку данных. Это могут быть мультипроцессорные системы с общей оперативной памятью, векторные процессоры, ассо-

циативные процессоры и др. Основная область применения параллельного программирования — решение сложных вычислительных задач. Основная цель — достигнуть максимального быстродействия машины. Характерной особенностью является то, что порождаемые процессы имеют, как правило, одну программу и действуют над данными одинаковой структуры. Например, каждый из них суммирует свою строку матрицы. Исследования в области параллельного программирования ведутся уже около тридцати лет, и здесь достигнут ряд успехов. Однако эта область лежит вне рассматриваемого ниже круга проблем. Для углубления в эту область можно посоветовать познакомиться с работами [16, 17].

3. *Конкурентное программирование* (термин «конкурентный» можно рассматривать как производный от слова «конкуренция»). В рамках конкурентного программирования программа, решающая некоторую конкретную задачу, представляется как совокупность множества процессов, выполняемых параллельно по отношению друг к другу. Программы этих процессов обычно различны. Здесь понятие параллельности в большой степени условно, так как если эти процессы выполняются на одной машине, то в каждый момент времени действительно выполняется процессором только один из них. В лучшем случае один процесс выполняется процессором, а один-два выполняют операции ввода-вывода. Иллюзия параллельности достигается за счет переключения процессора с одного процесса на другой в соответствии с некоторой *дисциплиной обслуживания*. Важно здесь то, что эти процессы выполняются на одной ЭВМ (возможно, на мультипроцессорной, но, как правило, на однопроцессорной) и в силу этого конкурируют между собой за ее физические или логические ресурсы: процессор, каналы ввода-вывода, области памяти, наборы данных и др. Конкурентное программирование возникло одновременно с появлением мультипрограммных операционных систем около двадцати лет назад, так как одновременное выполнение нескольких заданий под управлением такой ОС можно трактовать как выполнение нескольких конкурентных процессов. Но это так только с точки зрения ОС и ее разработчиков. С точки зрения программиста, выполняющего свою программу под управлением мультипрограммной ОС, она выполняется строго последовательно. Более того, обычно ОС не предоставляет в распоряжение программиста средств для явного взаимодействия процессов, выполняющихся в различных разделах памяти. Но вот мультизадачный режим работы (в терминологии ОС ЕС ЭВМ) — это уже настоящее конкурентное программирование в современном понимании, наиболее характерной особенностью которого является наличие у процессов возможности доступа к общей памяти. Несмотря на кажущуюся простоту обеспечения их взаимодействия — «В чем же проблема, если у процессов есть об-

щая память?» — конкурентное программирование требует строжайшей дисциплины и наличия соответствующих языковых средств. Те, кто пробовал писать и отлаживать мультизадачные программы (например, на языке Ассемблера или ПЛ/1 на ЕС ЭВМ), видимо, знают, как трудно обеспечить их надежность и как можно построить программу, которая, правильно проработав 99 раз, отказывает на сотый по совершенно непонятным причинам. При этом такие программы с трудом поддаются традиционным методам отладки, так как повторить ситуацию, приведшую к аварии, как правило, невозможно. Она может быть связана с соотношениями времен выполнения отдельных частей разных процессов, на которые к тому же могут влиять и внешние условия (прерывания выполнения более приоритетными задачами, занятость каналов ввода-вывода и др.). Повышению надежности конкурентных программ должны способствовать специальные конструкции применяемого языка программирования. Однако используемые в настоящее время языки либо вовсе не содержат этих средств, либо имеют крайне ограниченный и морально устаревший набор потенциально ненадежных средств, как, например, язык Ассемблера ЕС ЭВМ и ПЛ/1. Более современные языки, такие как Конкурентный Паскаль [18] и Ада [6], обладают, конечно, более совершенными средствами. В отличие от параллельного программирования, основная задача конкурентного программирования — обеспечение лучшей структуризации программы и иногда увеличение эффективности за счет большего совмещения работы процессора с работой каналов ввода-вывода. Улучшение структуризации достигается за счет того, что в ряде случаев разбиение программы на несколько независимых процессов упрощает ее логику и облегчает отладку и модификацию (при условии, конечно, что программа построена корректно с использованием надлежащих языковых средств). Выврожденными случаями конкурентных процессов являются сопрограммная структура программы и квазипараллельные процессы в языках имитационного моделирования. Интересно заметить, что в языках имитационного моделирования понятие процесса как отражение реальных процессов в имитируемых системах было четко сформулировано раньше, чем в языках программирования, еще в самом начале 60-х годов. Проблематика конкурентного программирования весьма и весьма близка к проблематике распределенного программирования. Многие задачи, требующие решения для обеспечения корректности программ, общие. Можно сказать, что распределенное программирование — это конкурентное программирование без допущения о наличии у процессов общей памяти.

4. *Распределенное программирование.* Определение его было дано выше. Заметим, что в распределенном программировании совсем не обязательно, чтобы процессы выполнялись на разных маши-

нах. Для того чтобы считать процессы распределенными и применять технику распределенного программирования, необходимо только, чтобы у них не было доступа к общей памяти. Таким образом, два процесса, выполняющиеся в разных разделах памяти мультипрограммной ОС, можно с успехом рассматривать, конечно, с точки зрения программиста как распределенные. Этим иногда пользуются при создании и отладке макета распределенной системы на одной машине, предполагая впоследствии перенести ее в реальную сеть.

Каждое из указанных выше направлений обладает своей спецификой и требует от языка программирования наличия в нем специальных средств, специфичных для данного круга задач, а от программиста — особого способа мышления, который во многом определяется имеющимися в его распоряжении языковыми средствами.

По аналогии с языками программирования низкого и высокого уровня эти средства можно подразделить на средства взаимодействия процессов низкого и высокого уровня. Первые обычно отражают в себе специфику используемого мультипроцессора (в случае параллельного программирования), операционной системы (в случае конкурентного программирования) или сети связи (в случае распределенного программирования). Как и языки ассемблеров, они обычно обеспечивают максимальную эффективность, но не слишком удобны в использовании и не обеспечивают переносимости программ. Примерами таких средств, видимо, наиболее близко знакомых по крайней мере части читателей, являются макрокоманды операционной системы. Например, в ОС ЕС есть так называемые макрокоманды супервизора для порождения, уничтожения и синхронизации подзадач (процессов). В сетях ЭВМ — это макрокоманды так называемых сетевых методов доступа, обеспечивающих пересылку сообщений от одного процесса к другому. Языковые средства взаимодействия процессов высокого уровня, как правило, независимы от специфики операционной среды, обеспечивают большее удобство программирования, повышенную надежность программ. Они лучше дисциплинируют мышление программиста, обеспечивая более совершенную структуризацию программы. Заметим, что далеко не всякие средства взаимодействия процессов, включенные в язык программирования высокого уровня, можно охарактеризовать как средства высокого уровня. Примером являются средства мультизадачности (асинхронного выполнения процедур) языка ПЛ/1. Они практически однозначно соответствуют низкоуровневым средствам супервизора операционной системы OS/360, одновременно с которой и для которой создавался язык.

Как уже было сказано, распределенное программирование (будем называть его для краткости РП-программированием) как самостоятельное направление развития языков программирования и

методологии программирования в целом сформировалось около десяти лет назад. Поэтому практически все разработанные языки РП-программирования еще не стали «промышленными» языками, находясь на стадиях реализации или экспериментальной эксплуатации. Надо сказать, что путь от идеи создания нового языка программирования до его применения в промышленном масштабе (когда на нем пишут тысячи и десятки тысяч программистов) чрезвычайно труден и долг. Полностью проходит его только один из нескольких сотен предлагаемых языков. Известно, что за всю историю вычислительной техники было разработано и реализовано несколько тысяч языков, но всего менее двадцати языков высокого уровня сейчас применяются достаточно широко. Многие удачные языки были разработаны, реализованы авторами, но применялись только в ограниченных масштабах в пяти — десяти организациях. Дело в том, что дать объективную оценку качества языка по сравнению с другими, предназначенными для решения того же класса задач, почти невозможно. Сложность в том, что качество языка может проявиться только во взаимодействии его с человеком, который пишет на нем и мыслит его понятиями. Известно, что индивидуальная производительность одного программиста может отличаться от производительности другого более чем на порядок. Поэтому на результаты каких бы то ни было экспериментов, которых, кстати, проводится не так уж и много, чрезвычайно сильно будут влиять индивидуальные способности участвующих в эксперименте лиц и даже их заинтересованность в том или ином результате эксперимента. На судьбу языка также сильно влияют такие чисто субъективные факторы, как известность его автора, солидность организации, разработавшей или поддерживающей язык, и даже соображения моды и престижности. Поэтому неудивительно, что до сих пор не существует общепринятого языка РП-программирования (не только у нас, но и за рубежом), что значительно облегчает нашу задачу выбора языка для иллюстрации приемов и методов РП-программирования. Средства РП-программирования языка АТ-Паскаль обладают достаточной для этой цели простотой, гибкостью и мощностью. Кроме того, использование того же языка, что и в предыдущих главах, позволит нам проиллюстрировать такие возможности, как использование аппарата АТД-программирования при построении распределенных систем и использование средств РП-программирования для реализации АТД. Таким образом, содержание этой главы можно рассматривать с двух точек зрения, либо как еще один пример применения техники АТД-программирования, либо как введение в технику РП-программирования с использованием АТД. Правда, следует признать, что, ввиду экспериментального характера современных средств РП-программирования и малой доступности для широких масс програм-

мистов сетей ЭВМ, эта книга, задуманная как практическое введение, приобретает здесь для большинства читателей слишком теоретический характер.

Может возникнуть вопрос: «Как же так? Сети ЭВМ существуют уже двадцать лет, а языков РП-программирования все еще нет?» Дело в том, что построение систем распределенной обработки данных — это далеко не единственный и не самый распространенный способ использования сетей ЭВМ. На первом месте здесь стоит доступ с терминалов одной подключенной к сети ЭВМ к интерактивным системам, выполняющимся на других ЭВМ (информационно-поисковым, разделения времени и т. п.). Многие потребности могут быть удовлетворены простой пересылкой файлов с машины на машину или доступом программы, выполняющейся на одной машине, к файлам или устройствам другой. При этом обработка данных остается централизованной, просто логически расширяется сеть устройств ввода-вывода, доступных процессам. Поэтому у многих пользователей и программистов, работающих с сетями, и не возникает потребности в каких бы то ни было специальных средствах программирования. Те же, кто разрабатывает сетевое программное обеспечение системного уровня (в частности, то, которое обеспечивает указанные выше режимы работы сети), как правило, довольствуются средствами РП-программирования низкого уровня, включенными в языки Ассемблеров. Эти средства, разумеется, возникли одновременно с сетями ЭВМ. Мы же говорим здесь о средствах РП-программирования высокого уровня, ориентированных на прикладных программистов.

В заключение следует сказать об одной важной и непривычной для программиста, использующего традиционные последовательные языки, особенности РП-программирования. В отличие от последовательного, параллельного и конкурентного программирования, при выполнении распределенных программ оказывается задействованным на порядок большее количество разнообразных аппаратных средств (ЭВМ, линии связи, адаптеры, связанные процессоры и др.). Суммарная надежность всей совокупности аппаратных средств резко падает. Поэтому идеальная распределенная система должна уметь надежно функционировать в условиях потенциально ненадежной аппаратной среды. Частично эта ненадежность ослабляется базовым сетевым программным обеспечением. Например, ошибки при передаче данных по линиям связи, особенно подверженным сбоям и помехам, обычно обрабатываются и устраняются связными процессорами с помощью повторных передач ошибочных сообщений. Тем не менее к программисту, разрабатывающему распределенную систему, часто предъявляется требование, чтобы система могла продолжать функционировать, хотя бы с частичной потерей своих функций, даже в случае выхода из строя одной или

нескольких машин. Рядовой программист, имеющий дело всегда с одной машиной, разрабатывает программу в предположении абсолютной надежности машины. В случае, если прогон его программы сорван из-за неисправности машины, ему остается только запустить свою программу на выполнение снова. В случае РП-программирования игнорировать ненадежность аппаратуры обычно нельзя. Однако вопрос защиты распределенной системы от сбоев аппаратуры настолько сложен, что требует отдельного детального рассмотрения. В нашем изложении мы постараемся не касаться его, оставаясь в рамках привычной «презумпции надежности» машины,

## 6.2. Средства РП-программирования языка АТ-Паскаль

Как и в случае средств АТД-программирования, мы начнем не с обзора средств РП-программирования, имеющихся в разных языках, а с одного конкретного варианта этих средств, включенных в язык АТ-Паскаль. Это тем более оправдано, что многие из ранее предложенных и используемых средств РП-программирования легко выражаются с помощью средств АТ-Паскаля, поэтому некоторые из них будут рассмотрены в качестве примеров применения этих средств.

Прежде чем говорить о языковых средствах, определим ту операционную среду, в которую будут у нас «погружены» процессы. В настоящее время применяется много различных видов сетей ЭВМ, которые отличаются архитектурой построения, топологией линий связи, принципами построения программного обеспечения и др. Чтобы не утонуть в массе деталей, мы сформулируем простую абстрактную модель сети, свойств которой должно быть достаточно для поддержки сетевых средств языка АТ-Паскаль, подобно тому, как для изучения программирования достаточно определить свойства некоторого абстрактного вычислителя и нет необходимости изучать все тонкости архитектуры машины и ее операционной системы.

Будем далее называть процессом выполнение последовательной программы на языке АТ-Паскаль. Эта программа отличается от обычной программы только тем, что в ее заголовке вместо зарезервированного слова PROGRAM используется слово PROCESS и тем, что в ней может использоваться специальная группа средств, которые мы и будем называть средствами РП-программирования языка АТ-Паскаль. Использование этих средств в обычной программе, начинающейся словом PROGRAM, недопустимо. Мы будем предполагать, что операционная среда обладает свойством порождать и уничтожать процессы. При порождении нового процесса ему при-

сваивается уникальное имя длиной не более восьми символов. Правила построения имен процессов зависят от конкретной сети связи. Либо это имя может быть чисто символическим, тогда сеть связи должна уметь по имени процесса определять его местонахождение, либо в имя может входить некоторая идентификация машины, на которой выполняется данный процесс. Мы будем использовать чисто символические имена. Заметим, что имя процесса не обязано совпадать с именем программы процесса, указанным в ее заголовке, но обычно мы будем предполагать, что эти имена одинаковы. Чтобы два процесса могли взаимодействовать, необходимо, чтобы по крайней мере один из них «знал» имя другого. Процессы могут обмениваться сообщениями, которые доставляются от одного процесса к другому и включаются в так называемую *очередь запросов*. Доставка сообщения выполняется асинхронно по отношению к выполнению передающего сообщения процесса. Это означает, что, послав сообщение, процесс не ждет какого бы то ни было подтверждения или ответа от процесса, принявшего его.

Рассмотрим теперь аппарат РП-программирования языка АТ-Паскаль. В современных языках РП-программирования сложились два основных подхода к обеспечению взаимодействия процессов: обмен сообщениями и удаленный вызов процедур. В случае обмена сообщениями основным *примитивом* (простейшей базовой операцией) является пересылка сообщения, т. е. *записи* в терминологии языка Паскаль, из памяти одного процесса в память другого процесса. Под синхронностью обмена сообщениями понимается то, что процесс, пославший сообщение другому процессу, задерживается до тех пор, пока не получит ответ на свое сообщение или служебное сообщение, подтверждающее, что отправленное им сообщение доставлено адресату. В случае асинхронного обмена сообщениями этой задержки не происходит. Под удаленным вызовом процедур понимается такой способ взаимодействия, когда один процесс может вызвать процедуру, принадлежащую другому процессу, т. е. процедуру, размещенную в тексте программы другого процесса. Обмен значениями входных и выходных параметров процедур обеспечивает необходимый при взаимодействии процессов обмен данными. Конкретные ограничения на виды и типы параметров и способы синхронизации процессов во время выполнения удаленного вызова могут варьироваться в разных реализациях этого аппарата. Обычно удаленный вызов процедур является синхронным, так как процесс, вызвавший некоторую удаленную процедуру, должен ожидать завершения ее выполнения и возврата значений выходных параметров. Краткий обзор известных языков РП-программирования высокого уровня будет дан ниже.

Основная идея аппарата взаимодействия процессов языка АТ-Паскаль, называемого *асинхронным удаленным вызовом процедур*,



состоит в том, что в качестве базового аппарата взят вызов процедуры, размещенной в другом процессе. Причем этот вызов выполняется асинхронно по отношению к вызывающему процессу. Это означает, что процесс, выдавший запрос на выполнение некоторой удаленной процедуры, продолжает свое функционирование, не дожидаясь ни начала, ни тем более конца ее выполнения. Из этого вытекает естественное ограничение, состоящее в том, что вызываемые так процедуры, называемые далее *глобальными*, так как их имена известны всем процессам, могут иметь только входные параметры. Передача данных при вызове глобальной процедуры происходит только в одну сторону — от процесса, вызывающего процедуру, к процессу, содержащему ее. Вызывающий процесс будем далее называть процессом-пользователем процедуры, а содержащий ее процесс — процессом-владельцем.

Вторая особенность данного аппарата в том, что действительное выполнение глобальной процедуры происходит не в тот момент, когда к процессу-владельцу пришел запрос на ее вызов, а когда владелец «дает разрешение» на ее выполнение. Это разрешение дается им в форме так называемого внутреннего вызова той же процедуры, на которую пришел запрос. Передачу запроса от пользователя к владельцу будем называть внешним вызовом. Для владельца процедура является точным аналогом обычной внутренней процедуры со своим списком параметров, который строится по обычным правилам языка АТ-Паскаль. Таким образом, глобальная процедура обладает двумя списками параметров: внешним, используемым при внешнем вызове процессом-пользователем, и внутренним, используемым при внутреннем вызове процессом-владельцем. Внутренний вызов может привести к задержке процесса-владельца, если внешний вызов данной процедуры еще не сделан или если имеющиеся вызовы не удовлетворяют некоторым условиям. Таким образом, внутренний вызов обладает синхронизирующими свойствами. Допускается также так называемый недетерминированный внутренний вызов, когда процесс делает одновременный внутренний вызов нескольких своих процедур. При этом выполняется только одна из них. Возможен также условный внутренний вызов, при котором выполнение процедуры происходит только тогда, когда есть подходящий запрос, а иначе процесс выполняет некоторое другое действие. Имеется также возможность задавать логическое условие выполнения процедуры в ее теле. При этом сама глобальная процедура может, проверив это условие, отменить свое выполнение.

Рассмотрим теперь синтаксические расширения языка АТ-Паскаль, необходимые для организации взаимодействия процессов. Объявление глобальной процедуры в процессе-владельце отличается от объявления обычной процедуры двумя моментами — заго-

ловком и так называемым условием входа. Заголовок глобальной процедуры имеет следующий синтаксис:

*заголовок-глобальной* = *внешний-заголовок*

[*внутренний-заголовок*].

*внешний-заголовок* = "GLOBAL" *заголовок-процедуры*.

*внутренний-заголовок* = "— >" [*имя*]

[(" *список-формальных-параметров* ")].

Внешний заголовок задает интерфейс процедуры, доступный процессам-пользователям. Список параметров в нем строится по обычным правилам языка со следующими исключениями. Вместо четырех видов параметров (константы, переменные, процедуры и функции) допускается только два (константы и процедуры). Типы параметров-констант могут быть любыми, за исключением ссылочных, а также регулярных и комбинированных со ссылочными компонентами. Это ограничение представится совершенно естественным, если вспомнить, что один процесс не имеет доступа к памяти другого, поэтому передавать ему адрес некоторой области памяти, определяемый значением ссылочного типа, бессмысленно. Если в списке параметров внешнего заголовка есть параметр-процедура, то предполагается, что соответствующий ему фактический параметр будет именем глобальной процедуры. Ограничения на список формальных параметров параметра-процедуры те же, что и у глобальной процедуры. Внутренний заголовок определяет интерфейс глобальной процедуры с процессом-владельцем. Заметим, что внутренний заголовок может определять и еще одно имя процедуры, которое используется владельцем. Это не является необходимым, но часто бывает удобным использовать различные имена глобальной процедуры для обеспечения большей мнемоничности. Никаких ограничений на виды и типы параметров внутреннего заголовка нет.

Раздел операторов глобальной процедуры может иметь вид "WHEN" *выражение составной-оператор*.

Логическое выражение, входящее в состав такого тела процедуры, определяет условие, при истинности которого процедура может быть выполнена. В выражение могут входить как переменные процесса-владельца, так и параметры процедуры (внешние и внутренние).

Любой процесс, осуществляющий внешний вызов глобальной процедуры, должен содержать в себе объявление ее внешнего интерфейса, имеющее следующий синтаксис:

*объявление-внешнего-интерфейса* =

*внешний-заголовок* ";"

"PROCESS" ("*имя-процесса*")" ";".

*имя-процесса* = *переменная* | *строка*.

Внешний заголовок должен совпадать с внешним заголовком этой процедуры в процессе-владельце. Директива PROCESS задает имя процесса-владельца данной процедуры. Оно может быть фиксированным (задаться строкой символов) или переменным (задаться переменной типа ALFA, значение которой должно быть присвоено до вызова данной глобальной процедуры).

Рассмотрим теперь синтаксис и правила выполнения внешнего и внутреннего вызовов. Внешний вызов строится по обычным правилам оператора вызова процедуры языка АТ-Паскаль. Выполнение этого оператора сводится к тому, что создается специальное сообщение, называемое далее *запросом на внешний вызов*. Это сообщение содержит имя глобальной процедуры, имя процесса и значения фактических параметров вызова. Сформированный запрос отсылается процессу-владельцу, а процесс-пользователь продолжает свое выполнение. Переданный по сети связи запрос включается в так называемую очередь запросов, которая имеется у каждого процесса. Запрос включается в очередь в момент его прихода, а выводится из очереди и уничтожается, когда требуемая процедура начинает выполняться. Запросы в очереди упорядочены по времени их поступления.

Оператор внутреннего вызова имеет довольно сложные синтаксис и семантику. Синтаксис его следующий:

*внутренний-вызов-глобальной-процедуры* =  
    *оператор-процедуры*  
    {"OR" *оператор-процедуры*}  
    ["ELSE" *альтернатива*];".

*альтернатива* = *оператор*.

В операторе может задаваться внутренний вызов либо одной глобальной процедуры (будем называть такой вызов простым), либо нескольких (такой вызов будем называть недетерминированным). Кроме того, может быть задана так называемая альтернатива, т. е. оператор, который выполняется в случае невозможности немедленного выполнения глобальной процедуры. Оператор внутреннего вызова с альтернативой будем называть условным, иначе — безусловным. Таким образом, имеются четыре модификации оператора:

- а) простой безусловный (A);
- б) простой условный (A ELSE S);
- в) недетерминированный безусловный (A OR B OR C);
- г) недетерминированный условный (A OR B OR C ELSE S).

Здесь A, B, C — вызовы процедур (возможно, с параметрами), S — произвольный оператор (в свою очередь может быть оператором внутреннего вызова).

Рассмотрим правила выполнения различных модификаций оператора.

а) Просматривается очередь запросов, в которой ищется первый запрос на данную процедуру. Из параметров запроса и параметров внутреннего вызова строится полный список параметров. Если в процедуре задано условие входа, то оно проверяется. В случае истинности условия (или если оно отсутствует) выполняется тело процедуры и на этом выполнение оператора внутреннего вызова завершается. Если условие входа ложно, то поиск в очереди продолжается.

Если вся очередь просмотрена и в ней не найден подходящий запрос, то процесс задерживается в операторе до прихода нового внешнего вызова. Затем выполнение оператора повторяется с начала.

б) Выполняется так же, как и безусловный, но если вся очередь просмотрена, процесс не задерживается. Вместо этого выполняется оператор S.

в) Выполняется так же, как простой безусловный, но в очереди ищется запрос на любую из указанных в операторе процедур. При выборе запроса порядок указания процедур в операторе внутреннего вызова не играет роли. Важен лишь порядок следования запросов в очереди.

г) Выполняется так же, как недетерминированный безусловный, но в случае, если ни одна из процедур не может быть выполнена немедленно, исполняется оператор S.

Рассмотрим действие оператора внутреннего вызова на простых примерах. Предположим, что некоторый процесс обладает глобальными процедурами следующего вида:

```
GLOBAL PROCEDURE A(K,L:INTEGER)
    —> (M,N:INTEGER);
    WHEN K=M
    BEGIN WRITE(L+N) END;
GLOBAL PROCEDURE B(K,L:INTEGER)
    —> (M,N:INTEGER);
    WHEN K<M
    BEGIN WRITE(L*N) END;
```

Предположим, что к некоторому моменту в очереди запросов стоят следующие запросы:

- 1) A(1,1);
- 2) A(3,2);
- 3) B(2,7);
- 4) A(2,4);
- 5) B(5,8);

Тогда результаты выполнения внутренних вызовов различных видов будут следующими:

<i>внутренний вызов</i>	<i>использованный запрос</i>	<i>результат</i>
A(1,2);	1	напечатано 3
B(3,3);	3	напечатано 21
A(6,8);	—	задержка
A(6,8)		
ELSE WRITE(0);	—	напечатано 0
A(3,3) OR A(2,3);	2	напечатано 5
A(6,8) OR B(6,8);	5	напечатано 64
A(4,7) OR B(4,7);	—	задержка
A(4,7) OR B(4,7)		
ELSE WRITE(0);	—	напечатано 0

Кроме описанных выше средств в язык АТ-Паскаль введены две новые стандартные функции:

1) REQCOUNT(P), где P — имя глобальной процедуры. Функция возвращает количество запросов к процедуре P во входной очереди. Может использоваться только в процессе-владельце процедуры.

2) USER. Функция возвращает имя процесса-пользователя процедуры. Может использоваться только в теле глобальной процедуры.

Заметим, что очередь запросов может быть реализована так, чтобы процессу была обеспечена возможность выполнения некоторых операций над ней. Например, в реализации языка АТ-Паскаль на ЕС ЭВМ все действия над очередью запросов выполняются написанными на языке АТ-Паскаль процедурами, оформленными в виде пакета, который автоматически присоединяется при компиляции программы с заголовком PROCESS. Очередь запросов представляется списком, реализованным с помощью пакета LIST, описанного выше. С помощью средств этого пакета процесс может просмотреть свою очередь запросов, переупорядочить ее, удалить ненужные запросы и т. п. Однако эти средства являются зависимыми от реализации языка, поэтому в дальнейшем мы ими пользоваться не будем.

Рассмотрим некоторые тонкости использования описанных выше средств. Как было сказано, на то, какой именно запрос будет выбран при выполнении недетерминированного оператора внутреннего вызова вида

A OR B OR C,

влияет только порядок запросов в очереди. Если необходимо обеспечить какой-либо определенный порядок просмотра очереди, например, чтобы сначала был обслужен запрос на процедуру A, если он есть, и только потом на B или C, можно воспользоваться тем, что альтернатива в условном операторе внутреннего вызова в свою

очередь может быть оператором внутреннего вызова. Например, оператор

A ELSE A OR B OR C

обеспечит требуемый порядок просмотра очереди. Сначала будет сделана попытка выполнить процедуру A, и только, если она не удастся, будут использованы запросы на B и C. Заметим, что оператор вида

A ELSE B OR C

неверен, так как если ни одна из процедур не сможет быть выполнена, то процесс перейдет в ожидание прихода запросов на B или C (но не на A). Приход запроса на процедуру A не выведет его из состояния ожидания.

Следует четко понимать, что условия входа проверяются только тогда, когда делается попытка выполнить глобальную процедуру при выполнении оператора внутреннего вызова. Никакие последующие изменения входящих в условия переменных, в том числе и такие, которые делают его истинным, не могут сами по себе инициировать выполнение процедуры. Например, если есть процедура

```
VAR V:INTEGER;  
GLOBAL PROCEDURE A( ... );  
WHEN V>0  
BECIN ... END;
```

то при значении  $V=0$  внутренний вызов процедуры не выполняется. Если затем переменная V получит значение 1, условие входа станет истинным, но тем не менее выполнение процедуры произойдет только тогда, когда для нее снова будет выполнен оператор внутреннего вызова.

Как было сказано, удаленный вызов глобальной процедуры обеспечивает только однонаправленный обмен данными. Как же обеспечить возврат результатов выполнения глобальной процедуры процессу-пользователю? Ответ очевиден. Если процесс P1 вызывает глобальную процедуру A в процессе P2, то после ее выполнения процесс P2 должен вызвать глобальную процедуру B в процессе P1, тем самым передавая ему результаты вызова процедуры A. Имя этой процедуры с ее списком параметров и имя процесса P1 либо могут быть известны процессу P2, либо он может вызывать процедуру, имя которой передано ему через параметры при вызове процедуры A. В этом случае процесс может даже не знать, какой процесс вызывал его глобальную процедуру. Такое на первый взгляд громоздкое решение оправдывают два обстоятельства. Во-первых, возврат результата выполнения глобальной процедуры требуется, как мы увидим ниже, далеко не всегда. В этих случаях

отказ от двунаправленности обмена при удаленном вызове обеспечивает возможность асинхронности. Во-вторых, возврат результата через обратный вызов может быть упрятан внутрь пакетов, реализующих те или иные виды взаимодействия процессов. При этом детали обмена данными будут вовсе не видны пользователю пакета,

### 6.3. Пакеты для РП-программирования

Достоинством описанного языкового аппарата является то, что он может использоваться и сам по себе как языковый аппарат достаточно высокого уровня, так и как средство для выражения с его помощью других известных средств взаимодействия процессов. Таким образом, аппарат АТ-Паскаля не заменяет, а дополняет их, что позволяет при построении распределенных систем на языке АТ-Паскаль использовать в полной мере весь ранее накопленный опыт, если он, конечно, есть у пользователя. Рассмотрим теперь несколько пакетов, определяющих три известных способа взаимодействия процессов. При этом мы преследуем три цели: дать примеры применения средств РП-программирования языка АТ-Паскаль, показать возможности использования аппарата пакетов в РП-программировании и продемонстрировать некоторые типичные виды взаимодействия процессов в распределенных системах.

Первая группа примеров связана с так называемой проблемой синхронизации процессов. Под синхронизацией в общем смысле можно понимать обеспечение ожидания одним процессом наступления некоторого события, связанного с деятельностью другого процесса. Например, процесс может ожидать, что другой процесс завершится, примет посланное ему сообщение, окончит работу с некоторым файлом и т. п. Исторически средства синхронизации процессов появились в языках программирования первыми. К ним относятся такие широко известные средства, как семафоры Э. Дейкстры [19], макрокоманды синхронизации подзадач в языке Ассемблера ЕС ЭВМ, события языка ПЛ/1 и др. Рассмотрим, как подобные средства могут быть определены с помощью аппарата удаленного вызова процедур. Легко видеть, что если некоторый процесс обладает глобальной процедурой следующего вида:

```
GLOBAL PROCEDURE SIGNAL —> WAIT;  
BEGIN END;
```

то внешний вызов этой процедуры (SIGNAL) будет не чем иным, как сигнализацией о наступлении некоторого события. Внутренний вызов (WAIT) будет обеспечивать ожидание процессом прихода этого сигнала. Процесс может, не переходя в состояние ожидания, проверить наличие сигнала о событии оператором условного внутреннего вызова вида

```
WAIT ELSE...
```

Такое решение может быть вполне приемлемым в простейшем случае, но в более сложном оно может оказаться непригодным. Если процессу должны сигнализировать разные процессы о наступлении различных событий, требуется столько процедур (с различными именами), сколько всего событий. Довольно легко при этом обеспечивается ожидание одного из нескольких событий, например, так:

#### WAIT1 OR WAIT2 OR WAIT3

(здесь WAIT1, WAIT2, WAIT3 — внутренние имена трех отдельных глобальных процедур). Однако обеспечить, например, ожидание любых двух из заданных трех событий невозможно. Поэтому усложним приведенную процедуру SIGNAL —> WAIT, добавив в ее заголовок параметры. Введем тип EVENT (событие), в качестве которого может быть взят любой скалярный тип. Значения этого типа будут идентифицировать некоторые события. Тогда процедура будет выглядеть так:

```
GLOBAL PROCEDURE SIGNAL(E:EVENT)
    —> WAIT(S:SET OF EVENT);
    WHEN E IN S
    BEGIN END;
```

Теперь одна процедура может «обслуживать» произвольное количество событий. Процесс, сигнализирующий о событии, указывает конкретное событие в параметре внешнего вызова, а процесс, ожидающий наступления событий, указывает множество событий, наступления одного из которых он будет ожидать. Например, если тип EVENT определен так:

```
TYPE EVENT = 1..13;
```

то процесс сигнализирует о наступлении события с номером 7 так

```
SIGNAL(7)
```

Если ждущий процесс должен ожидать одного из событий с номерами 5, 9, 10, 11, то он может сделать это с помощью внутреннего вызова вида

```
WAIT([5,9..11])
```

Легко сделать также, чтобы процесс мог указать, от какого именно процесса он ждет сигнала о событии. Для этого в список параметров внутреннего заголовка процедуры введем параметр P — имя процесса, а условие входа в процедуру запишем так:

```
WHEN (E IN S) AND (USER = P)
```



И, наконец, наиболее сложный случай, когда процесс хочет ожидать наступления  $N$  событий из заданного множества, содержащего  $M$  событий, где  $N$  меньше или равно  $M$ . При этом процесс должен получить информацию о том, какие именно события произошли. Сигнализация о событиях со стороны процессов-пользователей процедуры остается прежней. Процедура будет иметь следующий вид:

```
GLOBAL PROCEDURE SIGNAL(E:EVENT) —>
    WAIT(S:SET OF EVENT;
        N:INTEGER;
        VAR Q:SET OF EVENT);
    WHEN E IN S
    BEGIN
        N:=N-1;
        S:=S-[E];
        Q:=Q+[E];
        IF N > 0 THEN WAIT(S,N,Q)
    END;
```

Это пример рекурсивной глобальной процедуры. Ее тело содержит внутренний вызов самой этой процедуры. При каждом вызове переменная  $N$  уменьшается на единицу, а из множества  $S$  удаляется то событие, которое наступило. Так продолжается до тех пор, пока не наступит  $N$  событий. Заметим, что фактическому параметру, соответствующему формальному параметру  $Q$ , до вызова процедуры должно быть присвоено в качестве значения пустое множество. Вызов может иметь, например, такой вид:

```
EV:=[];
WAIT([5,9..11],2,EV)
```

Этот вызов обеспечит ожидание любых двух событий с номерами 5, 9, 10, 11. Значением переменной  $EV$  после вызова будет множество номеров совершившихся событий, например [5, 11].

Процедуру целесообразно оформить как пакет с заголовком

```
PACKAGE WAIT(EVENT:TYPE);
```

Необходим еще пакет `SIGNAL`, содержащий объявление глобальной удаленной процедуры `SIGNAL`. Этот пакет должен использоваться в процессах, посылающих сигналы о событиях. Он имеет вид

```
PACKAGE SIGNAL(EVENT:TYPE; P:INOUT);
GLOBAL PROCEDURE SIGNAL(E:EVENT); PROCESS(P);
```

Таким образом, эта пара пакетов определяет в совокупности некоторую форму взаимодействия процессов. Естественно, что объ-

явления типа EVENT должны быть одинаковыми во всех процессах, использующих эти пакеты. Если некоторый процесс (пусть его имя есть 'PROCESS7') должен иметь возможность ожидать событий с именами EV1, EV2, EV3, то тело процесса должно иметь раздел присоединения следующего вида:

```
USE WAIT((EV1, EV2, EV3));
```

(здесь (EV1, EV2, EV3) — объявление скалярного типа). Процесс может вызывать процедуру WAIT, например, так:

```
WAIT( [EV1, EV3], 1, [ ] )
```

Процесс, посылающий сигналы процессу 'PROCESS7', должен использовать раздел присоединения вида

```
CONST PR='PROCESS7';  
USE SIGNAL((EV1, EV2, EV3), PR);
```

и сигнализировать о наступлении события EV3 так:

```
SIGNAL(EV3)
```

Рассмотрим теперь вторую группу примеров, связанную с реализацией аппарата обмена сообщениями между процессами — первым языковым аппаратом, который стал применяться в распределенном программировании в первую очередь при разработке программного обеспечения сетей связи ЭВМ. Обмен сообщениями лежит в самом фундаменте распределенной обработки данных, так как аппаратура сетей связи ЭВМ действует именно на этом принципе. Сообщение является основной единицей обмена данными по линии связи. Во многих сетях передаваемое процессом сообщение разбивается на несколько более коротких сообщений, так называемых пакетов, которые передаются по сети более или менее независимо друг от друга и затем вновь собираются вместе (такие сети связи называются сетями с коммутацией пакетов). Несмотря на это, поскольку разборка сообщений на пакеты и их сборка производится программным обеспечением самой сети связи, основной единицей обмена данными между процессами с точки зрения разработчика прикладной распределенной системы все же остается сообщение. Как было сказано, асинхронный обмен сообщениями лежит и в основе реализации асинхронного удаленного вызова процедур в языке АТ-Паскаль. Интересно заметить, что, хотя аппарат обмена сообщениями возник из-за особенностей физической аппаратуры сетей связи, оказалось, что диктуемую им технику программирования удобно применять и при организации взаимодействия конкурентных процессов, особенно при построении операционных систем. Конечно, в этом случае никакой физической пересылки сообщения по линиям связи не происходит. Сообщение передается от процес-

са к процессу простой пересылкой его в памяти одной машины или даже пересылкой его адреса. В семидесятые годы остро стоял вопрос о том, какая схема взаимодействия процессов лучше: обмен сообщениями или вызов процедур (в случае сетей — удаленный вызов процедур). В конце концов пришли к соглашению, что обе схемы имеют равное право на существование. В одних случаях более удобна одна схема, в других — другая [20]. Более того, было показано [21], что обе эти схемы эквивалентны по своим возможностям. Программу, написанную в терминах одной из них, всегда можно переписать, используя другую. Кстати, еще одним подтверждением этого положения служат и средства языка АТ-Паскаль. В нем удаленный вызов процедур реализуется с помощью простого аппарата пересылки сообщений и в свою очередь служит средством реализации более сложных видов пересылки сообщений.

Рассмотрим теперь различные виды средств обмена сообщениями и их реализацию с помощью аппарата языка АТ-Паскаль. Для обозначения операций отправки и приема сообщения мы будем использовать общепринятые в РП-программировании обозначения SEND и RECEIVE соответственно. Процесс, передающий сообщение, будем называть *процессом-передатчиком* (SENDER) или для краткости просто передатчиком. Процесс, принимающий сообщение, назовем *процессом-приемником* (RECEIVER). Под сообщением будем понимать значение некоторого произвольного типа, который будем обозначать именем MESSAGE. Обычно — это запись или массив. Естественное ограничение состоит в том, что в сообщение не могут входить значения типа указатель, так как передача приемнику адреса участка памяти, к которой он не имеет доступа, бессмысленна.

Простейшая форма взаимодействия процессов — асинхронный обмен сообщениями. В этом случае процесс-передатчик не ждет ни ответа на посланное сообщение, ни подтверждения его приема процессом-приемником. Этот способ пересылки сообщений реализуется глобальной процедурой, принадлежащей приемнику, следующего вида:

```

— — ПЕРВЫЙ ВАРИАНТ ПРОЦЕДУРЫ SEND—>RECEIVE
GLOBAL PROCEDURE SEND(M:MESSAGE) —>
    RECEIVE( VAR N:MESSAGE );
BEGIN
    N := M
END;
```

Если Q — переменная типа MESSAGE, то отсылка сообщения передатчиком реализуется оператором

```
SEND(Q)
```

а прием сообщения приемником с возможным ожиданием его — оператором

## RECEIVE(Q)

Если процесс-передатчик не единственный, то приемнику может потребоваться информация о том, от какого именно процесса он получил сообщение, например, для того, чтобы впоследствии послать ему ответ. Этого легко достигнуть следующей процедурой:

```
— — ВТОРОЙ ВАРИАНТ ПРОЦЕДУРЫ SEND—>RECEIVE
GLOBAL PROCEDURE SEND(M:MESSAGE) —>
    RECEIVE(VAR N:MESSAGE;
            VAR FROM:ALFA);

BEGIN
    N := M;
    FROM := USER
END;
```

Может потребоваться не только это, но и то, чтобы приемник мог указать, от какого именно процесса он хочет принять сообщение в данный момент.

```
— — ТРЕТИЙ ВАРИАНТ ПРОЦЕДУРЫ SEND—>RECEIVE
GLOBAL PROCEDURE SEND(M:MESSAGE) —>
    RECEIVE(VAR N:MESSAGE; FROM:ALFA);
    WHEN USER = FROM
    BEGIN
        N := M;
    END;
```

Такое условие входа обеспечивает отбор из очереди только тех запросов, которые исходят от указанного параметром FROM процесса.

Часто применяется и другой принцип отбора поступивших сообщений — по имени порта. *Портом* называется логический канал связи процессов с другими. Понятие порта близко к понятию файла. Так же, как одновременно может быть открыто несколько входных файлов, и в каждой операции чтения указывается, из какого файла производится ввод, процесс может принимать сообщения из нескольких портов, и в каждой операции приема сообщения задается имя порта. Передатчик же всегда указывает, в какой порт приемника он посылает сообщение. Таким образом, каждый порт имеет свою логическую очередь сообщений. Мы говорим о логической очереди, так как на самом деле очередь запросов остается единственной. Пусть порты приемника обозначаются значениями некоторого типа PORT. Это могут быть символические имена, числа, строки, т. е. любые типы, над значениями которых допускается операция

сравнения. Тогда процедура SEND—>RECEIVE будет иметь следующий вид:

```
— — ЧЕТВЕРТЫЙ ВАРИАНТ ПРОЦЕДУРЫ SEND—>
    RECEIVE
GLOBAL PROCEDURE SEND(PS:PORT; M:MESSAGE)—>
    RECEIVE(PR:PORT; VAR N:MESSAGE);
    WHEN PS=PR
    BEGIN
        N := M;
    END;
```

Пусть, например, тип PORT объявлен так:

```
TYPE PORT=(PORT1, PORT2, PORT3);
```

тогда, если передатчик послал два сообщения операторами

```
SEND(PORT1,Q)
SEND(PORT2,S)
```

а приемник выполняет оператор

```
RECEIVE(PORT2,R)
```

будет принято сообщение S (Q, S, R — переменные типа MESSAGE).

Этот вариант процедуры SEND—>RECEIVE обеспечивает возможность введения приоритетности сообщений. Пусть, например, разные процессы-передатчики шлют сообщения в порты приемника: самые приоритетные в PORT1, менее приоритетные в PORT2, наименее приоритетные в PORT3. Тогда оператор

```
RECEIVE(PORT1,R)
ELSE RECEIVE(PORT2, R)
ELSE RECEIVE(PORT1,R)
OR RECEIVE(PORT2,R)
OR RECEIVE(PORT3,R)
```

обеспечит прием сообщений сначала из первого порта, если они есть, затем из второго порта и в последнюю очередь из третьего. Если ни в один из портов сообщение не поступило, то оператор обеспечит ожидание прихода сообщения в любой из них. При необходимости использовать этот оператор в разных местах программы можно выделить его в отдельную процедуру.

Любой из рассмотренных выше вариантов процедуры SEND—>RECEIVE можно оформить в виде пакета. Вернее, каждый из вариантов требует двух пакетов. Один из них будет содержать саму глобальную процедуру и использоваться в процессе-приемнике, а другой состоять из объявления удаленной глобальной процедуры и

использоваться в процессе-передатчике. Рассмотрим пример таких пакетов. Пусть пакет, содержащий процедуру передачи сообщения, называется SEND, а пакет с процедурой приема сообщения — RECEIVE. Мы возьмем для примера процедуру, аналогичную второму варианту приведенной выше процедуры SEND—>RECEIVE, возвращающей приемнику имя процесса, от которого принято сообщение. То, что в языке АТ-Паскаль имя процесса-владельца глобальной процедуры указывается не при вызове ее, а в объявлении,

```
PACKAGE SEND(MESSAGE:TYPE; . -- ТИП СООБЩЕНИЯ
      PARTNER:INOUT; -- ИМЯ ПРИЕМНИКА
      M      :INOUT); -- ПОСЫЛАЕМОЕ СООБЩЕНИЕ
PROCEDURE SEND( M:MESSAGE; PARTNER:ALFA );
GLOBAL PROCEDURE TRANSMIT( M:MESSAGE );

BEGIN
      TRANSMIT(M)
END

PACKAGE RECEIVE(MESSAGE:TYPE; -- ТИП СООБЩЕНИЯ
      PARTNER:INOUT; -- ИМЯ ПЕРЕДАТЧИКА
      N      :INOUT); -- ПРИНИМАЕМОЕ СООБЩЕНИЕ
GLOBAL PROCEDURE TRANSMIT( M:MESSAGE ) ->
      RECEIVE( VAR N:MESSAGE; VAR PARTNER:ALFA );
BEGIN
      N := M;
      PARTNER := USER
END;
```

Рис. 24. Пакеты SEND и RECEIVE

может оказаться неудобным, если процесс должен посылать сообщения разным процессам. Поэтому в пакете RECEIVE операция отсылки сообщения оформлена как локальная процедура, а не как глобальная. Имя процесса, которому посылается сообщение, задается в параметре процедуры SEND. Однако средства параметризации пакетов языка АТ-Паскаль позволяют сделать так, чтобы это имя можно было указать всего один раз при присоединении пакета, если передатчик работает с одним приемником. Аналогично, если в качестве передаваемого сообщения всегда фигурирует одна и та же переменная, ее имя можно задать как параметр при присоединении пакета.

Текст пакетов SEND и RECEIVE показан на рис. 24. Таким образом, чтобы некоторый процесс смог использовать операцию от

сылки сообщения некоторого типа, в его тексте должен быть присоединен пакет SEND. При этом тип MESSAGE должен быть либо объявлен в присоединяющем блоке, либо задан как фактический параметр пакета. Параметры PARTNER и M — необязательные. Если параметр PARTNER задан при присоединении пакета (его значением может быть имя переменной или константы типа ALFA), то указывать имя приемника в вызове процедуры SEND не надо. Если задано значение параметра M (им может быть имя переменной типа MESSAGE), то задавать имя переменной, в которой хранится сообщение, не надо. Таким образом, используя средства параметризации пакетов, мы построили процедуру отсылки сообщения с переменным количеством параметров. Аналогично используется и пакет RECEIVE. Тип MESSAGE должен быть объявлен в точности таким же, как и в процессе-передатчике. Некорректное объявление типа MESSAGE в приемнике или передатчике может вызвать ошибки при выполнении. К сожалению, поскольку программы процессов компилируются независимо друг от друга и, возможно, даже на разных машинах, у компилятора нет возможности проверить соответствие типов, как и при компиляции обычных раздельно компилируемых процедур. При выполнении процессов проверяется только соответствие общей длины всех параметров внешнего вызова объявлению внешнего заголовка глобальной процедуры. Если приемнику не нужны имена процессов, передающих ему сообщения, можно задать параметр PARTNER пакета RECEIVE (имя переменной типа ALFA). Тогда имя процесса-передатчика всегда будет присваиваться указанной переменной. Если задать фактический параметр N пакета, то принятые сообщения будут присваиваться этой переменной, и указывать соответствующий параметр в вызове процедуры RECEIVE будет не нужно. Рассмотрим примеры использования пакетов.

Пример 1.

— — ПРОЦЕСС—ПЕРЕДАТЧИК

USE SEND(ARRAY[1..100] OF INTEGER);

VAR A:MESSAGE;

SEND(A,'PROCESS7');

— — ПРОЦЕСС—ПРИЕМНИК

TYPE MESSAGE=ARRAY[1..100] OF INTEGER;

VAR PROCS:ALFA;

MES:MESSAGE;

USE RECEIVE(PROCS,MES);

RECEIVE;

— — СООБЩЕНИЕ ПОМЕЩАЕТСЯ В ПЕРЕМЕННУЮ MES

— — А ИМЯ ПРОЦЕССА—ПЕРЕДАТЧИКА—В PROCS

Пример 2.

— — ПРОЦЕСС—ПЕРЕДАТЧИК

```
USE STR=STRING(80);  
CONST PR='PROCESS7';  
USE SEND(STR,PR);  
VAR S:STR;
```

```
SEND(S);
```

— — ПРОЦЕСС—ПРИЕМНИК

```
USE STR=STRING(80);  
VAR PROCESSNAME:ALFA;  
VAR R:STR;  
USE RECEIVE(STR);
```

```
RECEIVE(R,PROCESSNAME);
```

Довольно часто взаимодействие процессов организуется таким образом, что процесс-передатчик, послав сообщение, должен ждать на него ответа от приемника. Ответ также является сообщением, в общем случае это сообщение может быть иного типа, чем исходное сообщение. Приемник же, получив сообщение, всегда должен ответить на него, послав ответное сообщение. Будем называть такую форму взаимодействия *посылкой сообщения с ожиданием ответа*. Разумеется, такую дисциплину обмена данными или, как принято называть это в сетях, *протокол* взаимодействия можно реализовать с помощью тех пакетов, которые мы уже рассмотрели. Однако более удобно определить специальный пакет, реализующий этот протокол. В нем будут уже не две, а четыре операции:

SEND—посылка исходного сообщения передатчиком;

RECEIVE—прием исходного сообщения приемником;

SENDREPLY—посылка ответа приемником;

WAITREPLY—ожидание ответа передатчиком.

При использовании этих операций процессу-приемнику не надо явно обрабатывать имена процессов-передатчиков. Операция SENDREPLY отсылает ответ тому процессу, от которого принято последнее сообщение. При выполнении операции WAITREPLY также проверяется, что ответ пришел именно от того процесса, которому послано последнее сообщение. Текст пакета, называемого SENDREPLY, приведен на рис. 25. Параметры пакета: типы MESSAGE и ANSWER (ответ) и P (необязательный) — имя процесса-приемника, если он единственный. Вот пример использования этого пакета:

— — ПРОЦЕСС—ПЕРЕДАТЧИК

```
USE MESSAGE=STRING(80);  
TYPE ANSWER=BOOLEAN;
```



```

VAR A:MESSAGE;
    B:ANSWER;
— — ИМЯ ПРОЦЕССА—ПРИЕМНИКА
CONST SERVER='SERVER ';
USE SENDREPLY;

. . .
SEND(A,SERVER);
WAITREPLY(B);
— — ПРОЦЕСС—ПРИЕМНИК
USE STR=STRING(80);
USE SENDREPLY(STR,BOOLEAN);
VAR S:STR;
    B:BOOLEAN;

. . .
RECEIVE(S);
. . . — — ВЫЧИСЛЕНИЕ ОТВЕТА
SENDREPLY(B);

```

Как видим, асинхронный обмен сообщениями предоставляет нам довольно большие возможности, однако иногда он оказывается непригодным. Рассмотрим, например, случай, когда процесс-передатчик шлет приемнику целый поток сообщений одно за другим. Эта ситуация может возникнуть, например, при пересылке целого файла. Если скорость обработки принимаемых приемником сообщений ниже, чем скорость их генерации передатчиком (при условии достаточно быстрых линий связи), принимаемые сообщения будут накапливаться в очереди запросов приемника, которая рано или поздно переполнится. Поэтому скорость работы передатчика должна быть согласована со скоростью работы приемника. Задача такого согласования часто называется задачей синхронизации (в узком смысле, в отличие от ранее упоминавшейся проблемы синхронизации в широком смысле).

Простейшим способом решения этой задачи является использование аппарата синхронного обмена сообщениями. При этом процесс, пославший сообщение, задерживается до тех пор, пока не получит уведомление о том, что посланное им сообщение принято. Здесь под приемом понимается не включение запроса на внешний вызов в очередь, а выборка запроса из очереди, т. е. выполнение соответствующей глобальной процедуры. Синхронность пересылки сообщений легко обеспечить, используя и асинхронный аппарат, если сразу после приема сообщения передатчик переходит в состояние ожидания некоторого события, а приемник, получив сообщение, сигнализирует об этом событии. Однако более удобно выполнять эти действия не в самих процессах, а включить их в процедуры передачи-приема сообщения. Такой вариант пакетов SEND и RECEI-

```

PACKAGE SENDREPLY( MESSAGE:TYPE;  -- ТИП СООБЩЕНИЯ
    ANSWER :TYPE;  -- ТИП СООБЩЕНИЯ-ОТВЕТА
    P      :INOUT); -- ИМЯ ПРИЕМНИКА
LOCAL VAR PARTNER:ALFA;
PROCEDURE SEND( M:MESSAGE; P:ALFA);
    GLOBAL PROCEDURE TRANSMIT( M:MESSAGE );
                                PROCESS(P);

BEGIN
    PARTNER := P; TRANSMIT(M)
END;
GLOBAL PROCEDURE TRANSMIT( M:MESSAGE ) ->
    RECEIVE( VAR N:MESSAGE );

BEGIN
    N := M; PARTNER := USER
END;
PROCEDURE REPLY( L:ANSWER );
    GLOBAL PROCEDURE SENDREPLY( L:ANSWER );
                                PROCESS(PARTNER);
BEGIN
    SENDREPLY(L)
END;
PROCEDURE SENDREPLY( L:ANSWER ) ->
    WAITREPLY( VAR K:ANSWER );
    WHEN USER = PARTNER
    BEGIN
        K := N
    END;

```

Рис. 25. Пакет SENDREPLY

VE показан на рис. 26. Заметим, что здесь при внешнем вызове глобальной процедуры TRANSMIT в списке ее параметров передается имя собственной глобальной процедуры WAIT. После выполнения отсылки сообщения процесс переходит на внутренний вызов этой процедуры и ждет, пока приемник не выполнит ее внешний вызов, сигнализируя тем самым о приеме сообщения. Заметим, что такое решение применимо для случая нескольких передатчиков и одного приемника. Вызов процедуры SIGNAL будет автоматически транслироваться процессу, вызвавшему глобальную процедуру TRANSMIT.

Для повышения скорости взаимодействия процессов иногда вводится требование, чтобы приемник обладал буфером фиксирован-

ного объема, в котором могло бы накапливаться W сообщений. В некоторых языках для управления этим буфером требуется порождение специального процесса. В языке АТ-Паскаль в случае одного передатчика и одного приемника никакого специального

--СИНХРОННЫЙ ВАРИАНТ ПАКЕТА SEND

```
PACKAGE SEND(MESSAGE:TYPE;    -- ТИП СООБЩЕНИЙ
              PARTNER:INOUT;   -- ИМЯ ПРИЕМНИКА
              M      :INOUT);  -- СООБЩЕНИЕ
PROCEDURE SEND( M:MESSAGE; PARTNER:ALFA );
GLOBAL PROCEDURE TRANSMIT( M:MESSAGE ;
                           PROCEDURE WAIT);
                           PROCESS(PARTNER)
GLOBAL PROCEDURE WAIT; BEGIN END;
BEGIN
    TRANSMIT(M,WAIT);
    WAIT
END
```

--СИНХРОННЫЙ ВАРИАНТ ПАКЕТА RECEIVE

```
PACKAGE RECEIVE(MESSAGE:TYPE; -- ТИП СООБЩЕНИЯ
                PARTNER:INOUT; -- ИМЯ ПЕРЕДАТЧИКА
                N      :INOUT); -- СООБЩЕНИЕ
GLOBAL PROCEDURE
    TRANSMIT( M:MESSAGE; PROCEDURE SIGNAL ) ->
    RECEIVE( VAR N:MESSAGE; VAR PARTNER:ALFA );
BEGIN
    SIGNAL;
    N := M;
    PARTNER := USER
END;
```

Рис. 26. Пакеты синхронной пересылки сообщений

процесса не требуется. Чтобы в очереди запросов всегда находилось не более чем W сообщений, приемник перед началом работы с передатчиком должен послать ему W фиктивных подтверждений. При этом передатчик может послать W сообщений, не дожидаясь пока приемник их получит, и только после этого перейдет в состояние ожидания. Пакет RECEIVE с ограниченным буфером получается из пакета RECEIVE на рис. 26 добавлением переменной

VAR W:INTEGER;

которой до первого вызова процедуры RECEIVE должно быть при-

своео значение — размер буфера. Оператор SIGNAL в процедуре RECEIVE заменяется на следующий оператор:

```
FOR I := 1 TO W DO SIGNAL; W := 1;
```

При этом после приема от передатчика первого сообщения ему посылается не одно, а W подтверждений, чем и разрешается послать

```
PACKAGE SENDFILE( F:INOUT;  -- ИМЯ ФАЙЛА
                  T:TYPE;    -- ТИП КОМПОНЕНТ ФАЙЛА
                  P:INOUT;    -- ИМЯ ПРОЦЕССА-ПРИЕМНИКА
                  W:CONST ); -- РАЗМЕР БУФЕРА
PROCEDURE SENDFILE( VAR F:FILE OF T; P:ALFA );
  USE WAIT([ACK]);
  GLOBAL PROCEDURE NEXTRECORD(
                  M:INTEGER; -- НОМЕР СООБЩЕНИЯ
                  X:T;       -- СООБЩЕНИЕ
                  E:BOOLEAN); -- ПРИЗНАК КОНЦА
                                PROCESS(P);

  VAR J,N:INTEGER;
  PROCEDURE SENDRECORD;
    VAR R:T;
    BEGIN -- SENDRECORD
      READ(F,R); N := N+1;
      NEXTRECORD(N,R,EOF(F));
      WAIT([ACK])
    END; -- SENDRECORD
  BEGIN -- SENDFILE
    RESET(F);
    N := 0;
    WHILE NOT EOF(F) DO SENDRECORD;
    FOR J := 1 TO W DO WAIT([ACK])
  END; --- SENDFILE
```

Рис. 27. Пакет пересылки файла

следующие W сообщений без ожидания. Дальше прием каждого сообщения подтверждает ровно один сигнал. Общее количество запросов, циркулирующих между процессами, не превышает W, чем и обеспечивается согласование скоростей передатчика и приемника.

В качестве последнего примера использования средств АТД-программирования в РП-программировании рассмотрим пакеты для пересылки файлов SENDFILE и RECVFILE (рис. 27, 28). Как

и в предыдущем примере, в них используется синхронный буферизованный обмен. Единственным обязательным параметром пакетов

```
PACKAGE RECVFILE( F:INOUT;  -- ИМЯ ФАЙЛА
                  T:TYPE;    -- ТИП КОМПОНЕНТ ФАЙЛА
                  P:INOUT;    -- ИМЯ ПРОЦЕССА-ПРИЕМНИКА
                  W:CONST ); -- РАЗМЕР БУФЕРА
PROCEDURE RECVFILE( VAR F:FILE OF T; P:ALFA );
  USE SIGNAL((ACK),P);
  VAR ENDFILE:BOOLEAN;
      J,N      :INTEGER;
  GLOBAL PROCEDURE NEXTRECORD( M:INTEGER;
                               X:T;
                               E:BOOLEAN );
  WHEN (M = N ) AND ( USER = P
  BEGIN -- NEXTRECORD
    SIGNAL(ACK);
    WRITE(F,X);
    N := N+1;
    ENDFILE := E
  END -- NEXTRECORD
  BEGIN -- RECVFILE
    FOR J := 1 TO W DO SIGNAL(ACK);
    N := 1;
    REWRITE(F);
    REPEAT NEXTRECORD UNTIL ENDFILE
  END; -- NEXTRECORD
```

Рис. 28. Пакет приема файла

является T — тип компонент файла (эти пакеты ориентированы на работу с нетекстовыми файлами). Если имя пересылаемого файла одно, его можно указать при присоединении пакета, иначе оно указывается при каждом вызове процедур. Имя процесса-партнера также может быть зафиксировано во время присоединения пакетов, иначе оно должно указываться в вызовах процедур. Таким образом, определенные в пакетах процедуры SENDFILE и RECVFILE являются процедурами с переменным количеством параметров. В зависимости от указанных при присоединении пакета параметров процедуры могут иметь один или два параметра или не иметь параметров вообще. Дополнительным свойством процедур, введенным для демонстрации возможностей условий входа, является контроль правильности следования компонент файла. Это может ока-

заться нужным, если сеть связи не гарантирует, что сообщение, посланное раньше другого, придет к получателю первым. Для этого при вызове процедуры пересылки одной компоненты (NEXTRECORD) вместе с компонентой передается ее порядковый номер. Условие входа

$M=N$

проверяет, что порядковый номер обрабатываемой записи совпадает с ожидаемым номером. Все сообщения с номерами, большими ожидаемого, будут задержаны в очереди до тех пор, пока не будет получено «запоздавшее» сообщение.

Рассмотрим пример использования этих пакетов. Предположим, что два процесса (PROC1 и PROC2) должны пересылать одно-типные файлы процессу CONSUMER.

```
— — ПРОЦЕСС PROC1
CONST CONSUMER='CONSUMER';
VAR W:FILE OF STRING;
USE SENDFILE(W,STRING,CONSUMER,5);

. . .
SENDFILE;
— — ПРОЦЕСС PROC2
CONST CONSUMER='CONSUMER'
VAR F:FILE OF STRING;
USE SENDFILE(,STRING,,5);

. . .
SENDFILE(F,CONSUMER)
— — ПРОЦЕСС CONSUMER
CONST PROC1='PROC1'; PROC2='PROC2';
VAR P,Q:FILE OF STRING;
USE RECVFILE(,STRING,,5);

. . .
RECVFILE(P,PROC1);
RECVFILE(Q,PROC2);
```

## 6.4. Примеры процессов для управления ресурсами

В предыдущем разделе мы рассмотрели ряд средств для обмена данными между процессами и решения задачи синхронизации. Другой широко известной задачей, решение которой часто требуется при организации взаимодействия процессов, является так называемая задача взаимного исключения процессов. Чтобы дать ее формулировку, необходимо ввести понятие ресурса. Под ресурсом мы будем понимать некоторую, обычно пассивную компоненту системы, используемую различными процессами. Разумеется, всегда имеются ресурсы, используемые одним и только одним процессом.

Такие ресурсы не представляют для нас интереса. Примерами ресурсов могут служить устройство ввода-вывода, область памяти, файл и даже процесс, с которым взаимодействуют другие процессы. Использование ресурсов процессами может быть различным. Некоторые ресурсы и некоторые виды их использования процессами могут допускать совместное использование ресурса несколькими процессами. Примером может служить общая область памяти (в случае конкурентных процессов), содержание которой процессы не изменяют. Другие ресурсы или другие способы их использования могут потребовать выделения ресурса одному процессу в монопольное владение на некоторое время. Например, если процесс хочет изменить содержимое общей области памяти, он должен сначала получить ее в монопольное владение, иначе содержимое общей области может стать некорректным. Поясним это на тривиальном примере. Допустим, что  $N$  — общая для двух процессов переменная (мы говорим здесь о конкурентных процессах). Время от времени процессы выполняют оператор  $N:=N+1$ . Переменная  $N$  является здесь общим ресурсом двух процессов. Пусть, например,  $N=0$ . Возможна ситуация (маловероятна, но возможна), что в тот момент, когда один процесс взял значение  $N$  для модификации, но еще не успел присвоить переменной новое значение, другой процесс также взял старое значение  $N$ . Затем оба процесса прибавили к старому значению, равному нулю, единицу и присвоили  $N$  полученные значения.  $N$  получит значение 1, хотя оператор  $N:=N+1$  был выполнен два раза, и  $N$  должно было бы получить значение 2. Значение  $N$  стало некорректным. Самое неприятное в этой ситуации то, что она крайне маловероятна, поэтому скорее всего, не будет выявлена на этапе отладки программы, а проявится рано или поздно во время ее эксплуатации. Причем, даже если она и возникнет один раз, повторить ее будет невозможно, и, скорее всего, ошибка не будет найдена вовсе. Вина будет списана на сбой машины, скачки напряжения питания, влияние биополей операторов или что-нибудь другое. Так и возникают ненадежные программные комплексы.

Конечно, в распределенных системах допустить такого рода некорректное обращение с общими ресурсами значительно труднее, чем в конкурентных, так как процессы не имеют общей памяти. Однако необходимость иметь общие для нескольких процессов ресурсы есть и в РП-программировании. Например, если разные процессы должны обращаться к некоторой таблице, хранящей нужную всем информацию, одной пересылкой сообщений не обойтись.

Наиболее известное решение задачи взаимного исключения в конкурентном программировании — это так называемые мониторы языка Конкурентный Паскаль [18]. *Монитор* — это языковая конструкция, близкая к понятию АТД. Монитор обладает набором переменных, недоступных процессам, значения которых отображают

текущее состояние ресурса. Для работы с ресурсом монитор имеет ряд процедур, доступных процессам. Основным свойством монитора является то, что в каждый момент времени процедуры монитора доступны только одному процессу. Если один процесс выполняет процедуру монитора, другие процессы не могут вызвать мониторные процедуры и встают в очередь к монитору. После окончания выполнения одного вызова мониторной процедуры монитор разрешает другому процессу вызвать процедуру, нужную ему.

Применение этой конструкции в РП-программировании тем более оправдано, что любой ресурс, представляемый некоторой структурой данных, может принадлежать одному и только одному процессу. Вводить специальные языковые расширения для реализации концепции монитора в языке АТ-Паскаль нет необходимости. В АТ-Паскале монитор — это специальный вид процесса. Действительно, процесс в языке АТ-Паскаль обладает всеми необходимыми свойствами монитора: он имеет собственные данные, представляющие текущее состояние ресурса, и набор доступных процессам процедур. Более того, если обычно дисциплина обслуживания запросов монитором однозначно определена в реализации языка, то в АТ-Паскале можно строить мониторы с любой дисциплиной обслуживания очереди запросов. Например, в зависимости от текущего состояния ресурса монитор может разрешать вызовы одних процедур и запрещать вызовы других. Удобным средством определения логики работы монитора могут быть и условия входов глобальных процедур. Итак, в языке АТ-Паскаль мы будем называть *процессом-монитором* или просто монитором такой процесс, который не выполняет никаких действий по своей инициативе, а занят исключительно обслуживанием других процессов, выполняющих вызовы его глобальных процедур. Простейший вид монитора — это процесс, имеющий, например, три процедуры А, В и С, тело которого есть оператор

```
WHILE TRUE DO A OR B OR C
```

или

```
LOOP A OR B OR C END
```

Обычно монитор работает в бесконечном цикле, однако можно предусмотреть в нем специальную процедуру для его остановки.

В качестве первого примера рассмотрим процесс-монитор, управляющий некоторым абстрактным ресурсом. Этот процесс не является в полном смысле монитором, так как он только дает разрешение на использование ресурса, например общего файла, а собственно доступ к нему процессы осуществляют самостоятельно. Будем предполагать, что ресурс допускает совместное использова-



ние. Процессы, использующие ресурс совместно, иногда называются процессами-«читателями», а использующие его монопольно — процессами-«писателями». Задача управления таким ресурсом иногда называется «проблемой читателей-писателей».

```

PROCESS RESOURCE;
  VAR R:INTEGER;  -- СОСТОЯНИЕ РЕСУРСА
  GLOBAL PROCEDURE WENQ( PROCEDURE SIGNAL );
  BEGIN
    SIGNAL;
    R := -1
  END;
  GLOBAL PROCEDURE RENQ( PROCEDURE SIGNAL );
  WHEN REQCOUNT(WENQ) = 0
  BEGIN
    SIGNAL;
    R := R+1;
  END;
  GLOBAL PROCEDURE DEQ;
  BEGIN
    R := ABS(R)-1
  END;
  BEGIN  -- ТЕЛО МОНИТОРА
    R := 0;
    WHILE TRUE DO
      CASE R OF
        -1:DEQ;
        0:WENQ OR RENQ;
        OTHERWISE DEQ OR RENQ
      END  -- CASE
    END.  -- МОНИТОР
  
```

Рис. 29. Монитор управления ресурсом

Текст процесса-монитора приведен на рис. 29. Он обладает тремя процедурами: RENQ — запрос на совместное использование ресурса, WENQ — запрос на монопольное владение ресурсом, DEQ — освобождение ресурса. Программистам, работавшим на языке Ассемблера ЕС ЭВМ, эти имена знакомы. Это имена макрокоманд управления ресурсами супервизора ОС ЕС ENQ и DEQ, которые выполняют те же функции, что и процедуры нашего монитора, хотя и более сложные. Состояние ресурса в мониторе изображается значением переменной R, R=0 означает, что ресурс свобо-

ден,  $R = -1$  — ресурс занят «писателем»,  $R = N$  — ресурс занимает совместно  $N$  «читателей». Логика управления ресурсом сосредоточена в теле монитора. При  $R = 0$  разрешаются вызовы  $RENQ$  и  $WENQ$ , т. е. захват ресурса любым процессом. При  $R = -1$  — только освобождение. При  $R > 0$  — освобождение и захват ресурса еще одним «читателем» при условии, что ни один «писатель» не ждет ресурса. Это предотвращает неопределенно долгое ожидание «писателей» в случае, когда поток «читателей» интенсивен. Кроме того, давать ресурс в распоряжение еще одного «читателя», видимо, не стоит, если в системе созрела необходимость модификации состояния ресурса каким-либо процессом. Текст программы монитора показывает, как процесс может управлять порядком вызова своих процедур. Заметим, что того же эффекта можно было бы достичь с помощью одних только условий входа. Для этого условие входа в процедуру  $WENQ$  должно быть следующим:

WHEN  $R = 0$

в процедуру  $RENQ$  таким:

WHEN ( $REQCOUNT(WENQ)$ ) AND ( $R \geq 0$ )

а в процедуру  $DEQ$ :

WHEN  $R < > 0$

Тело монитора в таком варианте будет следующим:

WHILE TRUE DO  $DEQ$  OR  $RENQ$  OR  $WENQ$

Поскольку обращаться к этому монитору будут различные процессы, целесообразно выделить все необходимые для доступа к нему объявления в один пакет (рис. 30). Процесс, желающий занять ресурс, должен сначала послать запрос на ресурс (вызов  $RENQ$  или  $WENQ$ ), а затем перейти в ожидание прихода сигнала о разрешении использовать ресурс. Для удобства и повышения надежности, чтобы пользователь пакета не мог забыть перевести процесс в ожидание, запрос к ресурсу делается в пакете локальной процедурой  $ENQ$  с параметром, равным  $RD$  (чтение) или  $WR$  (запись). Пример использования этого пакета (в предположении, что процесс использует два ресурса, управляемые мониторами  $RES1$  и  $RES2$ ):

USE  $R1 = RESOURCE('RES1')$ ,

$R2 = RESOURCE('RES2')$ ;

$R1.ENQ(R1.RD)$ ;

— — ИСПОЛЬЗОВАНИЕ РЕСУРСА  $RES1$

$R1.DEQ$ ;

$R2.ENQ(R2.WR)$ ;

— — МОДИФИКАЦИЯ РЕСУРСА  $RES2$

$R2.DEQ$ ;

Заметим, что с точки зрения программиста, разрабатывающего программу процесса, пакет RESOURCE выступает в полной мере как АТД-пакет, реализующий операции над некоторой абстрактной переменной типа RESOURCE. То, что реализация операций над ней выполняется другим процессом, и то, что с этой переменной может одновременно работать несколько процессов, скрыто от пользователя. Таким образом, эту пару — пакет RESOURCE и процесс RESOURCE можно рассматривать с двух точек зрения. Во-первых,

```
PACKAGE RESOURCE( MONITOR:CONST );
  TYPE MODE = (RD,WR);
  PROCEDURE ENQ( M:MODE );
    GLOBAL PROCEDURE WENQ( PROCEDURE W );
                                PROCESS(MONITOR);
  GLOBAL PROCEDURE RENQ( PROCEDURE W );
                                PROCESS(MONITOR);
  GLOBAL PROCEDURE WAIT; BEGIN END;
  BEGIN
    IF M = RD THEN RENQ(WAIT)
      ELSE WENQ(WAIT);
  WAIT
  END;
  GLOBAL PROCEDURE DEQ; PROCESS(MONITOR);
```

Рис. 30. Пакет для доступа к процессу RESOURCE

пакет можно рассматривать как набор интерфейсных процедур для обеспечения взаимодействия с процессом. Во-вторых, процесс может трактоваться как реализация АТД, спецификация которого определена пакетом. Таким образом, мы видим, что не только средства АТД-программирования могут пригодиться в РП-программировании, но и возможности, предоставляемые средствами РП-программирования, могут внести что-то новое в АТД-программирование, а именно реализацию АТД в виде независимых процессов.

Говоря о вопросах управления ресурсами, нельзя не остановиться на одной важной проблеме, с которой часто сталкивались разработчики конкурентных и распределенных систем (операционных систем и сетей ЭВМ). Это проблема тупиков. Предположим, что имеются два ресурса R1 и R2. Процесс P1 сначала захватывает ресурс R1, а процесс P2 — ресурс R2. Затем процесс P1 хочет занять ресурс R2, а процесс P2 — ресурс R1. Легко видеть, что оба процесса перейдут в состояние бесконечного ожидания. Действительно, P1 не может освободить R1, нужный для завершения P2, пока не получит ресурса R2, занятого процессом P2, который

не может освободить его, так как не может получить R1, занятый P1. Возникла ситуация тупика или *клинча*. Мы не будем здесь останавливаться на многих известных методах предотвращения тупиков, наша цель — только предостеречь читателя. Известно хотя и не самое эффективное, но простое решение, состоящее в том, что процессы должны захватывать все необходимые им для выполнения ресурсы сразу все вместе. При этом процесс ждет, пока все нужные ему ресурсы не станут одновременно свободными, и только после этого занимает их. Это легко реализовать, но потребуются монитор, управляющий всеми ресурсами распределенной системы. Очевидно, что такое «централизованное» управление в распределенной системе неэффективно. Второе очевидное решение состоит в запрете одновременного использования одним процессом сразу нескольких ресурсов, однако это не всегда возможно. Видимо, требуются некие компромиссные решения. Самое неприятное в этой проблеме то, что тупиковая ситуация может возникать с малой вероятностью. Поэтому ошибки в управлении ресурсами трудно уловимы при отладке.

Рассмотренный выше монитор RESOURCE не всегда применим в распределенной системе, так как он предполагает, что все процессы имеют физический доступ к управляемому ресурсу, что бывает в распределенных системах редко. Большее применение могут иметь мониторы, сами осуществляющие все операции над ресурсами. Пример такого монитора приведен на рис. 31. Он управляет общей для нескольких процессов таблицей. В данной версии таблица представлена массивом, хотя, разумеется, возможны и другие способы ее представления. Каждая строка таблицы является отдельным ресурсом, используемым совместно или монопольно. Для обеспечения управления доступом к строкам таблицы монитор имеет массив STATUS, каждый элемент которого отображает текущее состояние соответствующей строки таблицы. Значение FREE говорит о том, что строка *свободна*, т. е. доступна для чтения любому процессу. Если процесс взял значение строки для модификации, значением соответствующего элемента массива STATUS будет имя этого процесса. Занятая так строка таблицы не доступна никому до тех пор, пока занявший ее процесс не вернет в таблицу модифицированное значение строки. Процессы могут читать строки (процедурой READR), читать строки для последующей перезаписи их (процедурой READW) и перезаписывать ранее прочитанные строки (процедурой WRITEL). Процедура STOP останавливает работу монитора и записывает текущее состояние таблицы во внешний файл, из которого таблица считывается при следующем запуске монитора. Остановка монитора возможна только, когда завершены все ранее начатые операции по модификации строк таблицы. Для этого служит условие входа в процедуру STOP, использующее

```

PROCESS TABLE(F);
CONST N= . . .; -- РАЗМЕР ТАБЛИЦЫ
FREE='*****'; -- ПРИЗНАК СВОБОДНОГО ЭЛЕМЕНТА
TYPE ELEMENT= . . .; -- ТИП ЭЛЕМЕНТОВ ТАБЛИЦЫ
VAR A:ARRAY[1..N] OF ELEMENT; -- УПРАВЛЯЕМАЯ ТАБЛИЦА
STATUS:ARRAY[1..N] OF ALFA; -- КЕМ ЗАНЯТ ЭЛЕМЕНТ?
F:FILE OF ELEMENT; -- ФАЙЛ ДЛЯ СОХРАНЕНИЯ ТАБЛИЦЫ
I:1..N; M:INTEGER, -- M - СЧЕТЧИК МОДИФИКАЦИИ
GLOBAL PROCEDURE READR( J:1..N;
PROCEDURE SEND( B:ELEMENT ));
WHEN STATUS[J] = FREE
BEGIN SEND(A[J]) END;
GLOBAL PROCEDURE READW( J:1..N;
PROCEDURE SEND( B:ELEMENT ));
WHEN (STATUS[J] = FREE) AND (REQCOUNT(STOP) =-0)
BEGIN SEND(A[J]); STATUS[J] := USER; M := M+1 END;
GLOBAL PROCEDURE WRITEL( J:1..N; B:ELEMENT );
BEGIN
IF STATUS[J] <> USER THEN -- ДИАГНОСТИКА ОШИБКИ
. . .

ELSE BEGIN
A[J] := B; STATUS[J] := FREE; M := M-1
END
END;
GLOBAL PROCEDURE STOP;
VAR K:1..N;
WHEN M = 0
BEGIN
REWRITE(F); FOR K DO WRITE(F,A[K]); HALT
END;
BEGIN -- ТЕЛО ПРОЦЕССА
RESET(F); M := 0;
FOR I DO
BEGIN STATUS[I] := FREE; READ(F,A[I]) END;
WHILE TRUE DO
READR OR READW OR WRITEL OR STOP
END.

```

Рис. 31. Процесс для доступа к общему массиву

счетчик начатых модификаций (М). Когда принят вызов на процедуру STOP и монитор ждет, пока появится возможность завершиться, запрещается выполнение новых операций чтения для модификации. Это обеспечивается условием входа процедуры READW. Как и в предыдущем случае, этот процесс можно рассматривать как реализацию некоторого АТД, определенного соответствующим пакетом. Текст этого пакета не приводится, его разработка оставлена в качестве упражнения для читателя.

Как и обычные пакеты, АТД-пакеты, реализация которых осуществляется отдельным процессом-монитором, допускают взаимозаменяемость реализаций. Например, таблица в рассмотренном выше мониторе может быть представлена файлом с прямым доступом. Если обычно замена реализации АТД требует, как минимум, перекомпиляции программы, использующей пакет, в случае распределенной реализации АТД этого не требуется. Просто при запуске процессов, составляющих вместе некоторую распределенную систему, пользователь может запустить на выполнение тот или иной вариант монитора. Для отладки процессов, использующих монитор, несложно сделать пакет с тем же интерфейсом, что и пакет для доступа к монитору, но реализующий все необходимые операции над АТД локально. Это может значительно упростить отладку, позволяя отработать логику процесса, независимо от других процессов.

Третьим примером, который мы рассмотрим, будет монитор управления терминалом. Предположим, что несколько процессов, выполняющихся в общем случае на разных машинах, должны взаимодействовать с человеком-оператором, сидящим за терминалом, принадлежащим одной из машин. Процессы могут посылать на терминал одиночные информационные сообщения, не требующие ответа оператора, длиной не более 80 символов. Эти сообщения могут выводиться в произвольном порядке. При таком режиме работы терминал представляет собой совместно используемый ресурс. Если процесс хочет вывести на терминал текст, состоящий из нескольких строк, желательно, чтобы в него не включались сообщения от других процессов. Аналогично, если процесс хочет провести с оператором небольшой диалог — последовательность запросов процесса и ответов оператора, терминал должен быть временно закреплен за данным процессом до окончания диалога. В этих случаях терминал должен выступать в роли монопольно используемого ресурса.

Чтобы не вдаваться в технические детали обмена данными с терминалом, предположим, что ввод-вывод в локальном случае осуществляется двумя отдельно компилируемыми процедурами DISPREAD и DISPWRITE. С помощью этих процедур процесс-монитор TERMINAL (рис. 32) производит обмен с закрепленным

```

PROCESS TERMINAL;
CONST FREE = '*****'; -- ТЕРМИНАЛ СВОБОДЕН
                        -- ПРИ STATUS=FREE
TYPE LINE = PACKED ARRAY[1..80] OF CHAR;
VAR STATUS:ALFA;
PROCEDURE DISPWRITE( T:LINE ); EXTERN;
PROCEDURE DISPREAD( VAR T:LINE ); EXTERN;
GLOBAL PROCEDURE SEIZE( PROCEDURE SIGNAL );
    WHEN STATUS = FREE
    BEGIN STATUS := USER; SIGNAL END;
GLOBAL PROCEDURE RELEASE;
    WHEN STATUS = USER
    BEGIN STATUS := FREE END;
GLOBAL PROCEDURE DISPWR( T:LINE );
    WHEN (STATUS = USER) OR (STATUS = FREE)
    BEGIN DISPWRITE(T) END;
GLOBAL PROCEDURE DISPRD( PROCEDURE SEND( T:LINE ));
    VAR S:LINE;
    WHEN STATUS = USER
    BEGIN DISPREAD(S); SEND(S) END;
BEGIN -- ТЕЛО ПРОЦЕССА
    STATUS := FREE;
    WHILE TRUE DO
        IF STATUS = FREE THEN
            BEGIN DISPWR ELSE SEIZE OR DISPWR END
        ELSE
            BEGIN
                DISPWR ELSE DISPRD
            ELSE RELEASE OR DISPWR OR DISPRD
            END
        END
    END.

```

Рис. 32. Процесс управления терминалом

за ним устройством. Другие процессы для выполнения обмена с терминалом обращаются к монитору, через его глобальные процедуры:

DISPWR—вывод строки;

DISPRD—ввод строки;

SEIZE—захват терминала в монопольное использование;

RELEASE—освобождение ранее захваченного терминала.

Состояние ресурса (терминала) в мониторе отображается значением переменной STATUS. Если значение переменной есть константа FREE, терминал свободен. Он может служить для вывода одиночных сообщений или быть занят любым процессом. После выполнения процедуры SEIZE значение переменной STATUS становится равным имени захватившего терминал процесса. В этом состоянии монитор обслуживает только запросы от данного процесса.

```
PACKAGE TERMINAL(MONITOR:CONST);
TYPE LINE = PACKED ARRAY[1..80] OF CHAR;
PROCEDURE DISPSEIZE;
  GLOBAL PROCEDURE SEIZE( PROCEDURE A);
    PROCESS(MONITOR);
  GLOBAL PROCEDURE WAIT; BEGIN END;
  BEGIN
    SEIZE(WAIT); WAIT
  END;
PROCEDURE DISPWITE( T:LINE );
  GLOBAL PROCEDURE DISPWR( T:LINE );
    PROCESS(MONITOR);
  BEGIN DISPWR(T) END;
PROCEDURE DISPREAD( VAR L:LINE );
  GLOBAL PROCEDURE DISPRD( PROCEDURE B( T:LINE ));
    PROCESS(MONITOR);
  GLOBAL PROCEDURE RECEIVE ( T:LINE );
    BEGIN L := T END;
  BEGIN -- DISPREAD
    DISPRD(RECEIVE); RECEIVE
  END; -- DISPREAD
PROCEDURE DISPRLSE;
  GLOBAL PROCEDURE RELEASE; PROCESS(MONITOR);
  BEGIN
    RELEASE
  END;
```

Рис. 33. Пакет для доступа к процессу TERMINAL

Запросы от других процессов накапливаются в очереди запросов до тех пор, пока терминал не будет освобожден. Заметим, что тело монитора имеет довольно сложный вид, что отражает требования к порядку обслуживания запросов. Так, если терминал свободен, сначала обслуживаются все запросы на вывод одиночных сообщений и только после этого выполняется захват терминала каким-либо



процессом. Если терминал занят процессом, сначала обрабатываются запросы на вывод, затем на ввод (так как ввод ответа должен выполняться только после вывода на терминал вопроса) и в последнюю очередь на освобождение терминала.

Пакет для доступа к этому монитору, используемый в процессах-пользователях, показан на рис. 33. Он скрывает от программиста все объявления глобальных процедур, нужные для взаимодействия с монитором. Пакет определяет тип LINE (строка терминала) и четыре процедуры:

DISPWRITE—вывод строки на терминал;

DISPREAD—ввод строки с терминала;

DISPSEIZE—захват терминала;

DISPRLSE—освобождение терминала;

Заметим, что интерфейс первых двух процедур совпадает с интерфейсом процедур обмена в локальном случае. Это позволяет отлаживать работающий с терминалом процесс локально. Для этого необходим пакет, содержащий объявления процедур локального обмена с терминалом и фиктивные объявления процедур DISPSEIZE и DISPRLSE.

Рассмотрим пример использования пакета TERMINAL, предполагая, что процесс работает с двумя терминалами на разных машинах.

```
USE T1 = TERMINAL('TTY13  '),
```

```
    T2 = TERMINAL('TTY02  ');
```

```
VAR A,B:T1.LINE;
```

```
    P,Q:T2.LINE;
```

```
T1.DISPWRITE(A);
```

```
T1.DISPWRITE(B);
```

```
WITH T2 DO
```

```
BEGIN
```

```
    DISPSEIZE;
```

```
    DISPWRITE(P);
```

```
    DISPREAD(Q);
```

```
    DISPRLSE
```

```
END;
```

Заметим, что использовать переменные A и B при обращении к терминалу T2 нельзя, так как определенные в разных включениях пакета типа LINE неэквивалентны в смысле ссылочной эквивалентности, определенной в главе 2,

## ГЛАВА 7

### АТ-ПАСКАЛЬ, АДА И МОДУЛА-2

#### 7.1. Средства АТД-программирования языка Ада

Язык программирования Ада [6] является одним из наиболее перспективных языков программирования. В его разработке участвовало большое количество специалистов разных стран. Существует международный стандарт на язык Ада. Ведутся интенсивные работы по его реализации на различных типах ЭВМ, причем особое внимание уделяется надежности реализаций и соответствию их входных языков стандарту. Это должно обеспечить максимальную мобильность программ на языке Ада. Как известно, в прошлом, да и зачастую в настоящее время, различные реализации одного и того же языка, созданные независимо разными группами разработчиков, отличаются друг от друга. «Незначительные» изменения, сделанные в языке в процессе его реализации, иногда из стремления улучшить язык, а иногда из желания упростить реализацию, часто действительно незначительны, но тем не менее они могут значительно усложнить переход с одной реализации языка на другую, особенно при переходе на другой тип ЭВМ. Подход, принятый изначально в работах по реализации языка Ада, должен устранить такие неприятности.

Язык Ада принадлежит новому поколению языков программирования, разработанных на основе опыта, приобретенного разработчиками и пользователями языков в семидесятые годы. Большое влияние на разработку языка оказал язык Паскаль. Наиболее существенные отличия языка Ада от языка Паскаль заключаются в наличии в нем следующих средств:

- 1) аппарат параметризуемых и отдельно компилируемых пакетов, обеспечивающих развитые средства АТД-программирования;

- 2) аппарат взаимодействия конкурентных процессов достаточно высокого уровня;

3) аппарат обработки так называемых исключительных ситуаций, которые могут возникнуть во время работы программы (ошибочные ситуации и др.).

Техника АТД-программирования, которая описана выше и была проиллюстрирована примерами на языке АТ-Паскаль, весьма близка к той, которая используется в языке Ада. Некоторые идеи, например приватные типы, позаимствованы из языка Ада непосредственно. Автор надеется, что читатель, познакомившись с основными идеями АТД-программирования на примере языка АТ-Паскаль, без труда освоит языковой аппарат языка Ада, предназначенный для аналогичных целей.

Рассмотрим теперь, как же выглядят описанные выше средства АТД-программирования в языке Ада. Разумеется, этот обзор будет по необходимости кратким. Мы будем рассматривать средства АТД-программирования языка Ада в том же порядке, как описывались аналогичные средства языка АТ-Паскаль. В этом изложении мы будем следовать той версии языка, которая описана в [6], одной из лучших книг по языку Ада, вышедших в русском переводе. Хотя в процессе разработки в язык вводятся некоторые изменения, отчего эта версия может слегка отличаться от современного стандарта, мы будем пренебрегать этими несущественными синтаксическими различиями, так как в данном случае наша цель — дать всего лишь самое приближенное представление о возможностях языка Ада.

Основные предопределенные типы в языке Ада практически те же, что и в языке Паскаль. Это INTEGER, FLOAT (аналог REAL), BOOLEAN и CHARACTER. Можно определять так называемые перечисляемые типы, аналогичные скалярным типам языка Паскаль.

Переменные в языке Ада объявляются так же, как и в языке Паскаль. Например,

```
M,N:INTEGER;  
K,L:INTI;
```

Регулярные типы (массивы) строятся с помощью конструктора типа вида

```
ARRAY (тип-индекса) OF тип-компонент
```

Например,

```
TYPE INDEX IS RANGE 1..1000;  
TYPE MATRIX IS ARRAY(INDEX,INDEX) OF FLOAT;
```

В отличие от языка Паскаль можно при использовании регулярного типа, в объявлении которого указан не конкретный диапазон значений индекса, а имя типа, указывать конкретный диапазон значений индекса непосредственно при объявлении переменных.

Такие типы называются регулярными типами с неуточненными границами. Например,

```
A:MATRIX(1..10,1..10);  
B:MATRIX(1..5,1..200);
```

и даже

```
C:MATRIX (1..N,1..M); — где N,M — переменные.
```

Это позволяет в языке Ада работать с массивами переменного размера.

Строковый тип STRING определяется в так называемом предопределенном окружении как регулярный тип с неуточненными границами следующим образом:

```
TYPE NATURAL IS INTEGER RANGE 1..INTEGER'LAST;  
SUBTYPE STRING IS ARRAY(NATURAL) OF CHARACTER;
```

Это позволяет объявлять строки с заданными длинами

```
S1:STRING(1..80);  
S2:STRING(1..8);
```

Мы видим, что типы в языке Ада могут быть параметризуемыми, в то время как в языке Паскаль типы параметров не имеют. В языке АТ-Паскаль подобные средства обеспечиваются только аппаратом параметризуемых пакетов. Например, если пакет MATRIX имеет вид

```
PACKAGE MATRIX(IND1,IND2:TYPE);  
VAR MATRIX:ARRAY[IND1,IND2] OF REAL;
```

то его использование следующего вида:

```
USE A=MATRIX(1..10,1..10),  
B=MATRIX(1..5,1..200);
```

обеспечит объявление двух матриц А и В разного размера. Однако определять размер матрицы при выполнении программы, как это может быть сделано в языке Ада, тем не менее нельзя.

Итак, основные конструкторы типов языка Ада практически те же, что и языка Паскаль, за исключением конструктора множественного типа, который в языке Ада отсутствует.

Аппарата, аналогичного деструкторам типов языка АТ-Паскаль, в языке Ада нет, за исключением так называемого предопределенного атрибута типа BASE, обеспечивающего переход от так называемого подтипа к его базовому типу. Аналог функций-констант — это предопределенные атрибуты типов FIRST и LAST. Различие чисто синтаксическое. Вместо FIRST (A), LAST(A) пишется A'FIRST, A'LAST. Применительно к регулярному типу эти атрибуты изображают максимальное и минимальное значения индекса;

A'FIRST эквивалентно FIRST(INDEX(A[ J]));  
 A'FIRST(2) эквивалентно FIRST(INDEX(A[,J]));  
 A'FIRST(3) эквивалентно FIRST(INDEX(A[,J]));

и т. д.

В отличие от языка АТ-Паскаль, где полиморфная процедура определяется как пакет, содержащий только одну эту процедуру, в языке Ада аппарат конкретизации (параметризации на этапе компиляции) полиморфных процедур (называемых в языке Ада *родовыми* процедурами) хотя и идентичен аппарату параметризации пакетов, но формально считается отличным от него. Настраиваемыми при конкретизации полиморфной процедуры параметрами могут быть типы, переменные и имена процедур и функций. Имеется возможность задавать значения параметров по умолчанию. Разделения параметров на ключевые и позиционные нет. Каждый параметр может использоваться и как ключевой, и как позиционный, аналогично параметрам-типам и параметрам-константам языка АТ-Паскаль. Рассмотрим аналог полиморфной процедуры VECTSUM, приведенной в главе 2.

```

GENERIC( TYPE VECTOR;
         TYPE INDEX;
         TYPE ELEMENT IS INTEGER;
         FUNCTION '+'( X,Y:ELEMENT) RETURN ELEMENT
           IS VECTOR.'+' )
FUNCTION VECTSUM(A:VECTOR)
  RETURN ELEMENT IS
  S:ELEMENT;
  J:INDEX;
BEGIN
  S := A(A'FIRST);
  FOR J IN INDEX'SUCC(A'FIRST) .. A'LAST LOOP
    S := S+A(J);
  END LOOP;
  RETURN S;
END VECTSUM;
```

Мы видим, что отсутствие деструкторов типа заставляет при конкретизации процедуры, кроме типа VECTOR, определяющего тип массива, над которым действует процедура, передавать еще и тип его индекса, и тип его элементов. Для типа ELEMENT задано значение по умолчанию — INTEGER. Интересной особенностью языка Ада является то, что параметром полиморфной процедуры может быть операция (в данном случае операция '+'). Это значительно расширяет возможности процедуры, так как в языке Ада допускается переопределять знаки операций для вновь вводимых

типов данных. Например, если MATRIX — тип, определяющий вещественную матрицу, и имеется такой набор объявлений:

```
TYPE IND IS RANGE (1..100);  
      ARR IS ARRAY( IND ) OF MATRIX;  
FUNCTION '+'( X,Y:MATRIX ) RETURN MATRIX IS  
      . . .  
W:ARR;  
S:MATRIX;
```

то настройку процедуры, для того чтобы она могла выполнять операцию сложения над всеми элементами массива матриц, следует осуществить так:

```
FUNCTION MATRSUM IS NEW  
      VECTSUM( ARR,IND,MATRIX );
```

а вызов, как обычно:

```
S := MATRSUM(W);
```

Аналогичного расширения возможностей процедуры VECTSUM можно было бы добиться и в языке АТ-Паскаль, но для этого пришлось бы сделать параметром пакета имя процедуры сложения двух элементов вектора (ADD), а оператор

```
S := S+A[J];
```

заменить на

```
S := ADD(S,A[J]);
```

Однако это привело бы к резкому падению эффективности при использовании процедуры для тех типов, для которых операция сложения является предопределенной в языке (INTEGER, REAL, SET).

Заметим, что возможность переопределения в языке Ада имен операций распространяется и на имена процедур. Одновременно может быть объявлено несколько процедур с одним и тем же именем. Они должны различаться между собой количеством параметров или их типами, чтобы компилятор мог сделать правильный и однозначный выбор одной из них на основе информации о количестве и типах фактических параметров.

Понятие пакета в языке Ада близко к тому, которое используется в языке АТ-Паскаль, но есть и существенное различие. Пакет в языке Ада состоит обычно из двух частей: спецификации и тела. В спецификации объявляются константы, типы, переменные и заголовки процедур. Эти объекты (или, как их называют в языке Ада, ресурсы) доступны извне пакета. Тело пакета содержит реализацию процедур пакета и объявления локальных объектов,

недоступных пользователю пакета. Основное отличие пакета языка Ада от пакета языка АТ-Паскаль в том, что тело может компилироваться независимо от программы, использующей пакет. В языке АТ-Паскаль пакет также может содержать заголовки отдельно компилируемых процедур, но эти процедуры компилируются независимо одна от другой и не могут оперировать с локальными объектами пакета, что в ряде случаев оказывается неудобным.

Параметризация пакета в языке Ада осуществляется с помощью так называемого *родового заголовка*, аналогичного родовому заголовку полиморфной процедуры. Как и в случае полиморфных процедур, параметры пакетов бывают трех видов: параметры-переменные (фактическими параметрами могут быть и константы), параметры-типы и параметры-процедуры (и функции). Для всех видов пакетов могут быть заданы значения по умолчанию. Имена пакетов как внешние имена (в терминологии АТ-Паскаль), так и имена контекстов не могут быть параметрами пакета. Включение пакета в некоторую использующую его программу может быть осуществлено тремя путями. Доступ к ранее скомпилированным библиотечным пакетам осуществляется автоматически по указанию требуемых имен пакетов в заголовке компилируемого модуля. Текст пакета может быть непосредственно вложен в текст использующей его программы. Он может быть включен в текст из внешней библиотеки с помощью конструкции вида

PRAGMA INCLUDE( *внешнее-имя* );

В отличие от АТ-Паскаль, в языке Ада операции включения текста пакета из внешней библиотеки и его настройки на параметры разнесены. Настройка параметризуемого пакета (его конкретизация) осуществляется конструкцией следующего вида (мы используем терминологию АТ-Паскаль):

PACKAGE *имя-контекста* IS NEW *имя-пакета* (*параметры*);

При доступе к библиотечному пакету его имя указывается в так называемом *ограничении видимости сегмента компиляции* (процедуры, пакета или задачи). Это обеспечивает доступ к пакету так, как если бы данный сегмент компиляции был текстуально вложен в блок, содержащий спецификацию пакета. Заметим, что, когда мы говорим о включении текста пакетов из библиотеки или о доступе к библиотечному пакету, мы говорим только лишь о спецификации пакета. Реализация пакета к этому моменту может еще быть не определена.

Рассмотрим теперь, как в языке Ада обеспечивается доступ к объявленным в пакете объектам (для единообразия мы будем использовать введенный ранее термин *объекты пакета*, а не принятый в языке Ада — *ресурсы пакета*). Заметим сразу, что возможности,

аналогичной присоединению пакета в АТ-Паскаль, когда все объекты становятся непосредственно доступны, в языке Ада нет. Все пакеты языка Ада считаются вложенными.

Основной аппарат доступа к объектам пакета — уточненные имена, где в качестве имени контекста используется или имя, указанное в заголовке пакета (для непараметризуемых пакетов), или имя, данное пакету при конкретизации (для параметризуемых пакетов). Контекст пакета может быть полностью присоединен к контексту какого-либо внутреннего блока с помощью конструкции вида

USE *имя-контекста*;

аналогичной

USE CONTEXT ( *имя — контекста* );

языка АТ-Паскаль.

Присоединение контекста, распространяющееся только на один оператор, как это обеспечивается оператором присоединения WITH языка АТ-Паскаль, в языке Ада не разрешается. Аппарата, аналогичного аппарату пакетов-функций, в языке Ада нет. Нечто подобное допускается только лишь для полиморфных процедур и функций, но не для пакетов произвольного вида. Присоединение к контексту блока одного имени из контекста ранее определенного пакета осуществляется через переименование. В языке АТ-Паскаль мы можем написать

USE P.X;

сделав непосредственно доступным имя X из контекста P. В языке Ада аналогичное действие осуществляется с помощью объявления вида

X:INTEGER RENAMES P.X;

Объект X из контекста P (здесь — переменная типа INTEGER) станет непосредственно доступен под именем X (это имя может быть и другим). Аналогично могут переименовываться не только переменные, но и процедуры, модули (пакеты, задачи), исключительные ситуации, но не типы.

Для каждой процедуры, пакета или задачи может быть задано ограничение видимости. В отличие от языка АТ-Паскаль, где в ограничении видимости могут задаваться имена произвольных объектов, в языке Ада ограничение видимости оперирует только с именами модулей. Ограничить видимость только некоторыми заданными объектами данного пакета в языке Ада нельзя. В отличие от языка АТ-Паскаль, ограничение видимости языка Ада может не только сужать, но и расширять множество объектов, видимых



изнутри модуля, так как в нем могут указываться имена библиотечных пакетов.

Приватные типы языка Ада имеют тот же смысл, что и в АТ-Паскаль, вернее, идея приватных типов языка АТ-Паскаль позаимствована из языка Ада. В языке Ада существуют и так называемые ограниченные приватные типы, над которыми не разрешена операция присваивания. В спецификации пакета ограниченный приватный тип объявляется так:

**RESTRICTED TYPE A IS PRIVATE;**

Реализация приватного типа дается в так называемой приватной части спецификации пакета.

В явном объявлении локальных объектов в языке Ада необходимости нет. Если объект объявлен в спецификации пакета, он считается нелокальным, т. е. видимым извне пакета; если в теле — то он считается локальным.

Мы видим, что средства АТД-программирования языков Ада и АТ-Паскаль довольно близки. Две принципиально важные особенности аппарата языка Ада, отсутствующие в языке АТ-Паскаль, — это раздельная компиляция пакетов и переопределение имен процедур и обозначений операций. Первая из них имеет особенно большое значение при разработке крупных программных проектов (10—100 тысяч строк и более). Наиболее важной особенностью средств АТД-программирования языка АТ-Паскаль, отсутствующей в языке Ада, является аппарат деструкторов типов.

## **7.2. Средства взаимодействия процессов языка Ада**

Средства взаимодействия процессов языка Ада (называемых в нем *задачами*) ориентированы прежде всего на конкурентное программирование, хотя явного предположения о наличии у процессов общей памяти не делается. Имеется дополнительная возможность создавать процессы, оперирующие с общими переменными, хотя такой практики, видимо, следует избегать. Синтаксис объявления процесса в языке Ада весьма близок к синтаксису пакета. Как и в случае пакета, объявление задачи состоит из двух частей: спецификации, доступной другим задачам, и недоступного им тела задачи, которое может при необходимости компилироваться раздельно. В спецификации задачи перечисляются так называемые *входы* задачи — аналог удаленных процедур. Как и пакет, задача может определять свои типы, используемые в других задачах.

Аппарат вызовов входов является основным средством взаимодействия задач в языке Ада. Каждый вход идентифицируется именем и обладает списком параметров, полностью идентичным спис-

ку параметров процедуры. С точки зрения процессов-пользователей (здесь и далее мы будем использовать терминологию, введенную в языке АТ-Паскаль) вызов входа другой задачи является точным аналогом вызова процедуры. Вызов входа отличается от вызова глобальной процедуры языка АТ-Паскаль тем, что выполняется синхронно. Процесс-пользователь, выполняющий вызов входа другого процесса, задерживается в операторе вызова до тех пор, пока процесс-владелец входа не выполнит обработку вызова. Вторым отличием является то, что входы могут иметь не только входные, но и выходные параметры, поэтому обмен данными при вызове входа в языке Ада является двунаправленным. Входные параметры специфицируются зарезервированным словом IN, указываемым перед типом параметра, выходные — словом OUT.

При выполнении некоторым процессом вызова входа другого процесса запрос на вызов поступает в очередь запросов процесса-владельца. Этот запрос обрабатывается только тогда, когда процесс-владелец дает разрешение на его выполнение в форме так называемого оператора приема, в простейшем случае имеющего вид

ACCEPT *имя-входа (список-формальных-параметров)* DO  
    *группа-операторов*  
END;

При выполнении оператора приема очередной запрос на вызов удаляется из входной очереди, выполняется присваивание значений входным параметрам и исполняется заданная в операторе приема группа операторов. Во время ее исполнения могут быть присвоены значения выходным параметрам, которые по достижении конца оператора приема возвращаются процессу-пользователю. Последний, получив их, продолжает свое функционирование. Если запроса на данный вход в очереди нет, то процесс-владелец задерживается в операторе приема до поступления запроса на вход.

Если в языке АТ-Паскаль объявление глобальной процедуры и оператор внутреннего вызова, разрешающий ее выполнение, разделены, то в языке Ада эти две конструкции объединены вместе. Оператор приема является одновременно и «объявлением глобальной процедуры», и исполняемым оператором, разрешающим ее вызов. Таким образом, разные операторы приема, относящиеся к одному и тому же входу, могут специфицировать различные действия (списки параметров входа должны быть, конечно, одинаковыми и совпадать с объявлением входа в спецификации задачи). Совокупность этих языковых средств (оператора вызова входа и оператора приема входа) обеспечивает обмен данными между процессами и их синхронизацию. Принятая в языке Ада концепция взаимодействия процессов называется *концепцией randevu*, так как для выполнения взаимодействия должна произойти «встреча» двух

процессов: владельца входа, находящегося в операторе приема, и пользователя входа, находящегося в операторе вызова входа.

Для сравнения средств РП-программирования языков АТ-Паскаль и Ада рассмотрим простой пример — пересылку сообщения (без ответа) некоторого типа MESSAGE. Тип считается объявленным везде, где используется его имя (в АТ-Паскаль — в обоих процессах, в языке Ада — в спецификации задачи). Принятое сообщение помещается в переменную Z типа MESSAGE. В языке АТ-Паскаль это действие обеспечивается следующими объявлениями и операторами:

*процесс-передатчик:*

```
GLOBAL PROCEDURE TRANSMIT(A:MESSAGE);  
PROCESS(T);
```

```
...  
TRANSMIT(X); — — УДАЛЕННЫЙ ВЫЗОВ  
               — — ГЛОБАЛЬНОЙ ПРОЦЕДУРЫ
```

*процесс-приемник:*

```
GLOBAL PROCEDURE TRANSMIT(A:MESSAGE);  
BEGIN Z := A END;
```

```
...  
TRANSMIT; — — РАЗРЕШЕНИЕ НА ВЫПОЛНЕНИЕ  
            — — ГЛОБАЛЬНОЙ ПРОЦЕДУРЫ
```

В языке Ада это будет выглядеть так:

*спецификация задачи* (доступная процессу-пользователю):

```
TASK T IS  
ENTRY TRANSMIT(A:IN MESSAGE);
```

```
...  
END;
```

*процесс-передатчик:*

```
T.TRANSMIT(X);
```

*процесс-приемник* (задача T):

```
ACCEPT TRANSMIT(A:IN MESSAGE) DO  
Z := A  
END;
```

Предположим, что нам необходимо в разных местах процесса приемника записывать принятые сообщения в различные переменные. В языке АТ-Паскаль это реализуется следующим образом

```
GLOBAL PROCEDURE TRANSMIT(A:MESSAGE)  
—> RECEIVE(VAR Z:MESSAGE);  
BEGIN Z := A END;
```

```
...
```

RECEIVE(P);

RECEIVE(Q);

В языке Ада необходимо либо дублировать оператор приема

ACCEPT TRANSMIT(A:IN MESSAGE) DO

P := A

END;

ACCEPT TRANSMIT(A:IN MESSAGE) DO

Q := A

END;

либо объявить в теле задачи Т процедуру

PROCEDURE RECEIVE(Z:OUT MESSAGE) IS

BEGIN

ACCEPT TRANSMIT(A:IN MESSAGE) DO

Z := A

END;

END RECEIVE;

RECEIVE(P);

RECEIVE(Q);

Различие между языками АТ-Паскаль и Ада существует и в способе адресации процессов. В АТ-Паскаль объявление процедуры TRANSMIT позволяет передавать сообщения любому процессу. Это обеспечивается присвоением имени процесса в качестве значения переменной Т. В языке Ада имя задачи Т присутствует как бы в самом (уточненном) имени входа Т.TRANSMIT (X), поэтому при выполнении оно уже не может измениться. Существует аппарат порождения семейств однопоточных задач, где каждая конкретная задача идентифицируется некоторым индексом. Тогда возможен вызов входа вида

T(J).TRANSMIT(X);

Это обеспечивает вызов входа в J-й задаче семейства. Однако к данному семейству могут относиться только однопоточные задачи.

Для выполнения условного приема входа или выбора одного из нескольких имеющихся вызовов разных входов служит оператор отбора. Его общий вид таков:

SELECT

WHEN *условие 1* => *оператор приема 1*

OR WHEN *условие 2* => *оператор приема 2*

```

OR . . .
[ ELSE альтернатива ]
END SELECT;

```

При выполнении оператора отбора могут быть выполнены только те операторы приема, условия выполнения которых истинны (это аналогично условиям входа языка АТ-Паскаль). Выбор одного из нескольких имеющихся запросов производится недетерминированно. Если ни один оператор приема по каким-либо причинам не может быть выполнен (условия ложны или нет запросов в очереди), то выполняется альтернативное действие, если оно задано в операторе отбора, иначе процесс переводится в состояние ожидания до прихода новых запросов на так называемые *открытые* входы (для которых условия истинны). Вместо оператора приема в операторе отбора может стоять так называемый оператор задержки. В случае отсутствия запросов он обеспечивает ожидание процессом заданного интервала времени, а затем, если запрос так и не пришел, выполнение некоторого действия, например выхода из оператора отбора. Как видим, семантика оператора отбора языка Ада близка к семантике оператора внутреннего вызова АТ-Паскаль. Различия состоят в следующем:

1) В АТ-Паскаль нет аналога оператора задержки. При необходимости выполнить подобное действие (например, перейти на метку 1, если за время Т не придет ни одного запроса на вызов глобальных процедур А или В) в языке АТ-Паскаль надо воспользоваться процедурой, написанной на языке Ассемблера (пусть ее имя — DELAY). Оператор внутреннего вызова тогда будет выглядеть так:

```

A OR B FLSE
  BEGIN DELAY(T); A OR B ELSE GOTO 1 END;

```

2) Условия входов языка Ада не могут использовать значения параметров входов, поэтому отбор среди нескольких имеющихся вызовов входов в соответствии со значениями параметров невозможен. Аналогично, в условиях входов нельзя использовать имена процессов-пользователей.

Ограничение языка Ада, запрещающее использование в условиях входов параметров запросов и имен процессов-пользователей, принципиально. Оно проистекает, видимо, из того факта, что в концепции рандеву предполагается, что пересылка параметров входа происходит только тогда, когда начал выполняться оператор приема для данного вызова входа. До начала его выполнения, когда производится выбор вызова из очереди, значения параметров вызова еще не известны процессу-владельцу. В силу этого написать универсальные пакеты пересылки сообщений между произволь-

ными процессами, например, подобные тем, которые рассматривались в главе 6, в языке Ада нельзя. Вернее, создать средство можно, но это будет не пакет, а специальная задача. Эта задача будет выполнять коммутацию сообщений между произвольными процессами, принимая сообщения от любого процесса, записывая их в собственную очередь и пересылая эти сообщения адресатам по их запросам. Если учесть, что каждый вызов входа требует четырех взаимодействий процессов (процесс X посылает вызов входа процессу Y, Y возвращает процессу X разрешение на пересылку параметров, процесс X пересылает входные параметры, процесс Y возвращает выходные параметры), пересылка одного сообщения от процесса к процессу потребует восьми взаимодействий (пересылок по сети связи). Это может оказаться весьма неэффективным, особенно при использовании сети с низкоскоростными линиями связи. Именно поэтому выше отмечалось, что аппарат взаимодействия процессов языка Ада представляется малоприспособленным для РП-программирования. В случае же конкурентного программирования затраты могут оказаться вполне приемлемыми.

В некоторых случаях неудобства, вызываемые ограничениями на условия входов, можно обойти, используя так называемые семейства входов. Например, если необходимо, чтобы задача, принимающая сообщения, могла использовать несколько входных портов (очередей запросов), ее вход можно объявить так:

ENTRY TRANSMIT(1..N) (A:IN MESSAGE);

Тогда образуется семейство из N одинаковых входов. И в операторе вызова входа, и в операторе приема при этом требуется указывать индекс конкретного входа из данного семейства.

T.TRANSMIT(J) (X);

ACCEPT TRANSMIT(J) (A:IN MESSAGE) DO . .

Если каждый из процессов-передатчиков будет посылать сообщения в выделенный ему порт процесса-приемника, то последний сможет по желанию принимать сообщения от того или иного конкретного процесса. Вместе с тем в языке нет средств, позволяющих процессу принять вызов от любого из входов семейства или от заданного подмножества входов семейства.

### 7.3. Модули языка Модула-2

Как отмечает автор языка программирования Модула-2 профессор Н. Вирт: «Язык Модула-2 — потомок и прямой наследник языков Модула и Паскаль» [9]. В настоящее время язык Модула-2 рассматривается многими как непосредственная альтернатива языку

Паскаль. Сохранив все лучшие черты последнего, автор ввел в язык понятие модуля, непосредственно нацеленное на использование концепции абстрактных типов данных. Именно эти средства языка интересуют нас в первую очередь, их мы и рассмотрим.

Понятие модуля в языке Модуля-2 достаточно близко понятию пакета в языке АТ-Паскаль, но имеет и существенные отличия. Как и пакет, модуль может содержать в себе произвольные совокупности объявлений констант, типов, переменных и процедур, поэтому он непосредственно пригоден для определения АТД. Более того, модуль в языке Модуля-2 более четко отражает разделение АТД на видимый интерфейс и невидимую реализацию, так как может состоять из двух текстуально разделенных частей: *модуля определения* и *модуля реализации*. Модуль реализации может компилироваться отдельно от использующей модуль программы, которая в свою очередь способна компилироваться даже в том случае, когда модуль реализации еще не написан, достаточно наличия только модуля определения. Раздельная компиляция модулей с сохранением полного контроля типов делает язык Модуля-2 пригодным для построения программ сколь угодно большого размера.

Модули в языке Модуля-2 бывают двух типов: *локальные модули*, текстуально вложенные в использующую их программу, и *раздельно компилируемые модули*. Локальные модули наиболее близки пакетам языка АТ-Паскаль. Отличия их от пакетов состоят в основном в следующем:

1) Локальный модуль должен быть непосредственно включен в текст программы, процедуры или другого модуля, автоматического присоединения его из библиотеки не обеспечивается. Разумеется, отдельные реализации языка могут предусматривать внеязыковый аппарат типа оператора INCLUDE для включения в текст программы во время компиляции внешних текстов.

2) Каких бы то ни было средств параметризации модулей нет. Средства, аналогичные средствам декомпозиции типов языка АТ-Паскаль, также отсутствуют.

3) Все объявляемые в модуле объекты считаются локальными в нем и невидимыми снаружи модуля, кроме тех, которые явно объявлены экспортируемыми с помощью конструкции вида

EXPORT <список имен>;

4) Имеются только два вида доступа к объектам модуля: непосредственный доступ по имени и доступ по уточненному (в смысле языка АТ-Паскаль) имени. Доступ по уточненному имени должен применяться, если имена экспортируемых объектов в модуле объявляются конструкцией вида

EXPORT QUALIFIED <список имен>;

5) Имена из контекста, объемлющего модуль, невидимы в нем, за исключением тех, которые явно указаны в так называемом списке импорта вида

```
IMPORT <список имен>;
```

6) Аналогом приватных типов являются так называемые типы со скрытым экспортом. Однако скрытый экспорт возможен только в раздельно компилируемых модулях и только для ссылочных типов.

7) Кроме наборов объявлений модуль имеет тело. *Тело модуля* выполняется перед телом объемлющей его процедуры или программы и может использоваться, например, для присваивания начальных значений переменным модуля.

Для раздельно компилируемых модулей правила видимости имен несколько иные, так как для них не существует объемлющего контекста. Все раздельно компилируемые модули существуют как бы на одном уровне с главной программой. Объявленные в них объекты существуют на протяжении всего времени выполнения программы. Как уже говорилось, модуль разделяется на модуль определения и модуль реализации. Имена объектов, объявленных в модуле определения, потенциально доступны из других модулей, которые должны содержать список импорта вида

```
IMPORT <имя модуля>;
```

который делает доступными по уточненным именам все объекты, объявленные в модуле определения. Список импорта вида

```
FROM <имя модуля> IMPORT <список имен>;
```

делает указанные в нем имена доступными непосредственно. Таким образом, список импорта делает доступным во время компиляции модуль определения некоторого модуля и вызывает подключение к программе модуля реализации при редактировании связей.

Мы видим, что язык Модуля-2 обладает весьма лаконичным набором средств АТД-программирования. Вместе с тем эти средства прекрасно соответствуют требованиям метода пошаговой реорганизации. Недостатком их, на наш взгляд, является отсутствие аппарата параметризации модулей, что затрудняет определение параметризуемых АТД. Если в локальных модулях возможна имитация параметризации с помощью импорта имен из объемлющего контекста, то в раздельно компилируемых модулях это недопустимо. Ведь в них список импорта должен содержать имя модуля, из которого импортируются «параметры», но при написании модуля оно не может фиксироваться, так как библиотечные модули могут использоваться в различных программах. Кроме того, раздельная компиляция модулей сама по себе чрезвычайно затрудняет параметри-



вацию модулей. В языке АТ-Паскаль развитый аппарат параметризации пакетов удалось ввести именно за счет отказа от раздельной компиляции, что, конечно же, несколько ограничило максимальный размер обрабатываемых программ.

Заметим, что в языке Модуля-2 аппарат определения непараметризуемых АТД не только обеспечивает пользователям потенциальную возможность применения методики АТД-программирования, но и интенсивно используется при определении самого языка. Так, например, набор стандартных процедур языка минимален и содержит в основном процедуры, реализуемые *in-line*. Все процедуры ввода-вывода, например, определены в библиотечных модулях. Таким образом, в языке Модуля-2 тип *файл* является не предопределенным типом языка, а в самом строгом смысле абстрактным типом. Аналогично определяются средства управления памятью и другие.

Средств РП-программирования язык Модуля-2 не имеет. Средства взаимодействия процессов в нем ограничены порождением и взаимодействием сопрограмм и обработкой прерываний, т. е. ориентированы прежде всего на персональные ЭВМ и мини-ЭВМ, не имеющие мультипрограммных операционных систем,

## П Р И Л О Ж Е Н И Е

### ФОРМАЛЬНОЕ ОПИСАНИЕ ЯЗЫКА АТ-ПАСКАЛЬ

#### 1. Способ определения синтаксиса

Это приложение служит справочным пособием для тех, кто желает углубиться в синтаксические тонкости языка АТ-Паскаль. В приложении дается полное описание синтаксиса языка. Семантика языка рассматривается в самой минимальной степени. Автор рассчитывает на то, что читатель, которого заинтересовало это приложение, знаком с языком Паскаль. Общеизвестные понятия и конструкции, общие для всех диалектов языка Паскаль, никак не поясняются и не иллюстрируются примерами в данном описании. Семантика новых введенных в Паскаль средств описана в главах 2, 3 и 6.

Для описания синтаксиса языка АТ-Паскаль используются расширенные формулы Бэкуса—Наура, аналогичные тем, которые применены при определении международного стандарта языка Паскаль [7]. В формулах метасимвол  $=$  отделяет левую часть формулы (определяемое понятие) от правой части (определения). Метасимвол разделяет альтернативные варианты определения. Квадратными скобками выделяются конструкции, которые могут быть опущены. Фигурными скобками  $\{ \}$  выделяются конструкции, которые могут быть повторены нуль или более раз. Конструкция вида  $(A|B|C)$  определяет выбор одной из альтернатив. Точка завершает формулу. Названия синтаксических элементов языка (нетерминальные символы) записываются строчными буквами без какого бы то ни было выделения их. В нетерминальных символах, состоящих из нескольких слов, эти слова разделяются дефисами. Терминальные символы заключаются в двойные кавычки.

#### 2. Алфавит языка

основной-символ=буква | цифра | специальный символ.  
буква=заглавная буква латинского или русского алфавита,  
цифра=десять арабских цифр.

специальный символ =

"+"	"-"	"*"	"/"	"="	"<="
">="	"("	)"	"<"	">"	"(*"
*)"	"::="	"."	","	"@"	"<>"
"["	"]"	"**"	;"	":"	"'"
".."	"(#"	"#)"	"— —"	"—>"	

зарезервированное-слово.

зарезервированное-слово =

"AND"	"ARRAY"	"BEGIN"	"CASE"
"CONST"	"DIV"	"DO"	"DOWNT0"
"ELSE"	"END"	"FILE"	"FOR"
"FORALL"	"FUNCTION"	"GLOBAL"	"GOTO"
"IF"	"IN"	"INITIALVALUES"	
"INOUT"	"LABEL"	"LOCAL"	"LOOP"
"MOD"	"NIL"	"NOT"	"OF"
"OR"	"OUT"	"PACKAGE"	"PACKED"
"PRIVATE"	"POSTLUDE"	"PROCEDURE"	"PROCESS"
"PROGRAM"	"RECORD"	"RESTRICTED"	"REPEAT"
"SET"	"THEN"	"TO"	"TYPE"
"UNTIL"	"USE"	"VAR"	"WHEN"
"WHILE"	"WITH".		

В тексте программы могут быть комментарии двух видов:

1. (\* комментарий\*)

2.— комментарий

Комментарии второго вида заканчиваются концом строки текста. Они могут встречаться либо в конце строки, либо занимать отдельную строку.

Пробелы, комментарии и концы строк текста программы являются разделителями. Разделители не могут встречаться внутри числовых и строковых констант, имен и зарезервированных слов. По крайней мере один разделитель должен отделять друг от друга идущие подряд константы, имена или зарезервированные слова.

В языке АТ-Паскаль разрешается использование альтернативных изображений некоторых специальных символов:

альтернативный символ	вместо
(.	[
.)	]
&	AND
	OR
⌋	NOT
⌋ =	<>

### 3. Числа, строки и имена

число-без-знака =

целое-без-знака |

вещественное-без-знака.

целое = последовательность.

последовательность = цифра {цифра}.

вещественное-без-знака =

последовательность "." последовательность |

последовательность "." последовательность

"E" порядок |

целое "E" порядок.

порядок = целое | знак целое.

знак = "+" | "-".

строка = "" символ {символ} "".

имя = буква {(буква | цифра)}.

уточненное-имя = имя-контекста "." имя | имя.

имя-контекста = уточненное-имя.

Пр и м е р ы уточненных имен: S.R P.R.X

### 4. Определение констант

Определение константы представляет имя как синоним константы. Значения компонент структурных констант (записей или массивов) перечисляются в порядке следования компонент записей или массивов соответствующих типов.

определение-константы =

имя "=" (константа | константа-множество) |

имя "=" структурная-константа ":" тип.

константа =

число-без-знака | знак число-без-знака |

имя-константы | знак имя-константы |

строка | "NIL".

имя-константы = уточненное-имя.

константа-множество = "[" "список-элементов-множества" "]".

список-элементов-множества = пусто |

элемент-множества {"", " элемент-множества}.

элемент-множества = константа [".." константа].

структурная-константа =

"(#" (константа | константа-множество)

{"", " (константа | константа-множество) " #)"

## 5. Определение типа данных

Определение типа присваивает ему собственное имя, которое может использоваться в программе для ссылок на этот тип.

определение-типа = имя "=" тип.

тип = имя-типа | конструктор-типа | деструктор-типа.

конструктор-типа = простой-тип | составной-тип |  
ссылочный-тип.

имя-типа = уточненное-имя.

### 5.1. Простые типы.

простой-тип = скалярный-тип | ограниченный-тип.

скалярный-тип =

"REAL" | "INTEGER" | "CHAR" |

"BOOLEAN" |

(" имя {", " имя} ").

ограниченный-тип = константа ".." константа.

### 5.2. Составные типы.

составной-тип = ["PACKED"] (регулярный-тип |  
комбинированный-тип | множественный-тип |  
файловый-тип).

регулярный-тип =

"ARRAY" "[" тип-индекса {", " тип-индекса} "]"

"OF" тип-компонент.

тип-индекса = простой-тип | деструктор-типа.

тип-компонент = тип.

комбинированный-тип =

"RECORD" список-полей [ "; " ] END.

список-полей = общая-часть [ вариантная-часть ] |  
вариантная-часть.

общая-часть = раздел-записи { "; " раздел-записи }.

раздел-записи = имя-поля { ", " имя-поля } ":" тип.

вариантная-часть =

"CASE" [ дискриминант ":" ] тип-дискриминанта

"OF" вариант { ", " вариант }.

вариант = список-меток-варианта ":"

(" [список-полей [ "; " ] ] ").

список-меток-варианта =

метка-варианта { ", " метка-варианта }.

метка-варианта = константа.

дискриминант = имя.

тип-дискриминанта = простой-тип. (\* кроме REAL \*)

имя-поля = имя.

множественный-тип = "SET" "OF" базовый-тип.

базовый-тип = простой-тип | деструктор-типа.

Примечание. Если базовым типом множества является тип CHAR, элементами множества могут быть только латинские буквы, цифры и специальные символы. Русские буквы не могут быть элементами множества.

файловый-тип = "FILE" "OF" тип.

Примечание. В языке определен стандартный файловый тип TEXT. Файлы типа TEXT состоят из последовательности символов, разбитой на отдельные строки. Предопределенные в языке файлы INPUT и OUTPUT являются текстовыми файлами.

ссылочный тип = "@" имя-типа | "REF" "(" тип ")".

Примечание. Первая форма конструктора ссылочного типа допускает использование ссылки на тип, который будет определен позднее. Вторая форма такой возможности не допускает, однако разрешает использовать не имя типа, а непосредственно конструктор или деструктор типа.

деструктор-типа =

имя-деструктора "(" аргумент-деструктора ")".

имя-деструктора = имя.

аргумент-деструктора =

деструктор-типа |

уточненное-имя |

компонента-аргумента.

компонента-аргумента =

аргумент-деструктора селектор-аргумента.

селектор-аргумента =

селектор-массива |

селектор-поля |

селектор-указателя |

селектор-параметра.

селектор-массива = "[" { "," } "]"

селектор-поля = "." имя-поля.

селектор-указателя = "@"

селектор-параметра = "(" { "," } ")".

Примечание. Определено четыре деструктора: TYP, INDEX, ELEM, BASE, их значение описано в главе 2.

### 5.3. Идентичность типов.

В связи с введением в язык АТ-Паскаль средств декомпозиции типов, понятие идентичности типов в нем изменено по сравне-

нию со стандартной версией языка, где используется так называемая именная идентичность. Именная идентичность является частным случаем принятой в языке АТ-Паскаль ссылочной идентичности, которая подчиняется следующим правилам:

1. Каждое имя константы, типа, переменной, функции, параметра процедуры или функции обладает так называемой *ссылкой на тип*, однозначно идентифицирующей тип, которому этот объект принадлежит.

2. Имена стандартных типов обладают уникальными ссылками на типы.

3. При определении нового типа (при вычислении конструктора типа) создается новый тип и конструктор возвращает ссылку на него.

4. Объявление вида

TYPE имя = тип;

присваивает имени ссылку на новый тип.

5. Объявление вида

TYPE имя-типа-1 = имя-типа-2;

присваивает имени-типа-1 ссылку на тот тип, на который ссылается имя-типа-2.

6. Объявления вида

VAR X1, ..., Xn:тип;

присваивает всем объявляемым именам одну и ту же ссылку на тип.

7. Деструкторы типов, выделяющие составные части ранее определенных типов, возвращают ссылки на эти составные части.

8. Два имени считаются принадлежащими одному и тому же типу (типы имен считаются идентичными) в том и только в том случае, когда они обладают ссылками на один и тот же тип.

5.4. Совместимость типов.

Приватные типы совместимы только в случае их идентичности.

Не приватные типы T1 и T2 считаются совместимыми, если выполняется одно из следующих соотношений:

— тип T1 идентичен типу T2;

— один из них является ограниченным типом, для которого другой является базовым типом;

— оба типа являются ограниченными по отношению к одному и тому же базовому типу;

— оба типа являются множественными типами с одним и тем же базовым типом;

— оба типа есть типы PACKED ARRAY [1..N] OF CHAR, то есть являются строками одинаковой длины.

5.5. Совместимость для присваивания.

Приватные типы совместимы для присваивания только в случае их идентичности,

Не приватный тип T2 совместим для присваивания с типом T1 (значение E типа T2 может быть присвоено переменной типа T1), если выполняется одно из следующих соотношений:

- типы T1 и T2 идентичны и не являются файловыми типами;
- T1 есть тип REAL, а T2 совместим с типом INTEGER;
- T1 и T2 есть совместимые нестандартные скалярные типы, а значение выражения E лежит внутри диапазона значений типа T1;
- T1 и T2 есть совместимые множественные типы и все элементы множества E лежат внутри диапазона, определяемого базовым типом типа T1;
- T1 и T2 являются совместимыми строковыми типами,

## 6. Объявления и обозначения переменных

объявление-переменной = имя { ", " имя } "; " тип.

переменная =

полная-переменная |  
компонента-переменной |  
указанная-переменная.

полная-переменная = имя-переменной.

имя-переменной = уточненное-имя.

**Примечание.** В левой части оператора присваивания в теле функции в качестве полной переменной может использоваться имя объявляемой функции. В структурных функциях за ним может следовать имя поля или список индексных выражений,

компонента-переменной =

индексированная-переменная |  
обозначение-поля |  
буфер-файла.

индексированная-переменная =

переменная-массив

"[" выражение { ", " выражение } "]"

переменная-массив = переменная | обозначение-функции.

**Примеры:** A[J], M[ I+J, K ], STRFUN1(A,B) [I,J]

обозначение-поля = переменная-запись "." имя-поля.

переменная-запись = переменная | обозначение-функции.

имя-поля = имя.

**Примеры:** R.P, Q[K].W, STRFUN2(A,B).FIELD

буфер-файла = переменная-файл "@".

переменная-файл = переменная.

указанная-переменная = переменная-ссылка "@".

переменная-ссылка = переменная,



**Примечание.** В языке АТ-Паскаль, в отличие от стандартного Паскаля, если в индексированную переменную входит не переменная-массив, а переменная типа *ссылка на массив*, то автоматически производится неявный переход от этой ссылки к массиву, на которую она указывает. Например, если есть такие объявления:

```
TYPE V = ARRAY[1..M] OF REAL;  
VAR A:ARRAY[1..N] OF REF(V);
```

то выражения  $A[I][J]$  и  $A[I, J]$  эквивалентны  $A[I]@[J]$ , единственно правильному в стандартной версии языка. Аналогичные действия производятся, если в обозначение поля записи входит не переменная-запись, а переменная типа *ссылка на запись*. Например,

```
TYPE R = RECORD X,Y,Z:REAL END;  
VAR S:REF(R);
```

Выражения  $S@.X$  и  $S.X$  эквивалентны. Заметим, что такая неявная отсылка к массиву или записи производится только в этих двух случаях. Выражения  $S@$  и  $S$  неэквивалентны,

## 7. Инициализация переменных

Начальные значения могут быть присвоены только переменным, объявленным в самом внешнем блоке программы. В качестве начальных значений массивов и записей используются структурные константы.

инициализация-переменной =

полная-переменная " := " константа-или-множество |

полная-переменная " := " структурная-константа,

## 8. Выражения

константа-без-знака = число-без-знака | "NIL" |

имя-константы | строка | функция-константа.

множитель = тело-множителя |

тело-множителя операция-степени множитель.

тело-множителя = константа-без-знака | переменная |

обозначение-функции | множество |

"(" выражение ")" | "NOT" множитель.

множество = "[" список-элементов "]".

список-элементов = элемент { ",", элемент } |

пусто.

элемент = выражение [ ".." выражение ].

слагаемое = множитель |

слагаемое мультипликативная-операция множитель.

простое-выражение = слагаемое | знак слагаемое |  
 простое-выражение аддитивная-операция слагаемое.  
 выражение = простое-выражение |  
 простое-выражение операция-отношения простое-выражение,

### 8.1. О п е р а ц и и.

операция-степени = "\*\*".  
 мультипликативная-операция =  
 "\*" | "/" | "DIV" |  
 "MOD" | "AND".  
 аддитивная-операция = "+" | "-" | "OR".  
 операция-отношения =  
 "=" | "<" | ">" | "<=" | ">=" |  
 ">" | "<" | "IN".

П р и м е ч а н и е. IN—операция проверки членства во мно-  
 жестве. Операция DIV—деление нацело.

### 8.2. О б о з н а ч е н и я   ф у н к ц и й.

обозначение-функции = имя-функции |  
 имя-функции "(" фактический-параметр  
 { "," фактический-параметр } ")".  
 имя-функции = уточненное-имя.  
 фактический-параметр =  
 выражение | переменная |  
 имя-процедуры | имя-функции.

### 8.3. Ф у н к ц и и - к о н с т а н т ы.

функция-константа = имя "(" аргумент-деструктора ")".

Функции-константы являются разновидностью деструкторов типа. Их аргументы строятся по правилам аргументов деструкто-  
 ров, а значениями являются константы. Определено три функции-  
 константы: SIZE, FIRST, LAST. Их смысл пояснен в главе 2,

## 9. О п е р а т о р ы

оператор =  
 [ метка ":" ] ( простой-оператор |  
 сложный-оператор ).  
 метка = целое-без-знака.  
 простой-оператор =  
 оператор-присваивания |  
 оператор-процедуры |  
 оператор-перехода |  
 событие |

пустой-оператор.  
 пустой-оператор = пусто.  
 оператор-присваивания = переменная ":"=" выражение  
 имя-объявляемой-функции ":"=" выражение.  
 имя-объявляемой-функции = имя.  
 оператор-процедуры = имя-процедуры |  
 имя-процедуры "(" фактический-параметр  
 {" фактический-параметр } ").  
 имя-процедуры = уточненное-имя,  
 оператор-перехода =  
 "GOTO" метка |  
 "GOTO" имя-константы.  
 сложный-оператор =  
 составной-оператор |  
 выбирающий-оператор |  
 оператор-цикла |  
 внутренний-вызов-глобальной-процедуры |  
 оператор-присоединения.  
 составной-оператор = "BEGIN" оператор  
 {" оператор } "END".  
 выбирающий-оператор =  
 условный-оператор | оператор-варианта .  
 условный-оператор =  
 "IF" выражение "THEN" оператор  
 ["ELSE" оператор]  
 оператор-варианта =  
 "CASE" выражение  
 "OF" элемент-списка-вариантов  
 {" ; " элемент-списка-вариантов }  
 [ [ " ; " ] "OTHERWISE" оператор  
 {" ; " оператор }  
 {" ; " ] ]  
 "END" .  
 элемент-списка-вариантов =  
 список-меток-варианта ":"оператор  
 список-меток-варианта =  
 метка-варианта {" метка-варианта } .  
 метка-варианта = константа [".." константа] .  
 оператор-цикла =  
 цикл-с-предусловием |  
 цикл-с-постусловием |  
 цикл-с-параметром |  
 цикл-по-множеству |

цикл-до-события .  
 цикл-с-предусловием =  
     "WHILE" выражение "DO" оператор .  
 цикл-с-постусловием =  
     "REPEAT" оператор  
     {";" оператор}  
     "UNTIL" выражение .  
 цикл-с-параметром =  
     "FOR" параметр-цикла [":=" список-цикла]  
     {";" параметр-цикла [":=" список-цикла] }  
     "DO" оператор .  
 список-цикла = выражение ("TO" | "DOWNTO") выражение .  
 параметр-цикла = полная-переменная .

**Примечание.** В отличие от стандартной версии языка в языке АТ-Паскаль можно не задавать начальное и конечное значение параметра цикла. В этом случае параметр цикла «пробегают» все множество значений, предопределяемое его типом, т. е. если тип I есть 1..100, то оператор

FOR I DO . . .

переберет все значения от 1 до 100. Оператор

FOR I, J, K DO . . .

эквивалентен

FOR I DO FOR J DO FOR K DO . . .

цикл-по-множеству =

"FORALL" параметр-цикла "IN" выражение  
 "DO" оператор .

**Примечание.** Параметр цикла пробегает все значения, которые являются элементами множества, определенного выражением.

**Пример:** FORALL J IN [ 1, 3, 5, 17..22 ] DO W[J] := 0  
 цикл-до-события =

"LOOP" оператор {";" оператор} "END" |  
 "LOOP" "UNTIL" событие {";" событие} ":"  
     оператор {";" оператор}  
 "POSTLUDE" событие ":" оператор  
     {";" событие ":" оператор}  
 "END" .

событие = имя .

**Примечание.** Цикл выполняется до тех пор, пока в теле цикла не будет выполнен оператор-событие. Есть предопределенное

имя события EXIT, которое приводит к выходу из цикла. Во второй форме оператора цикла конструкция POSTLUDE определяет действия, которые выполняются после выхода из тела цикла в зависимости от того, какое именно событие привело к этому.

оператор-присоединения =

"WITH" список-присоединения "DO" оператор .

список-присоединения =

элемент-списка-присоединения

{", " элемент-списка-присоединения} .

элемент-списка-присоединения =

переменная запись | имя-контекста.

**Примечание.** В отличие от стандартной версии языка Паскаль, в языке АТ-Паскаль в операторе присоединения допускается указывать имя контекста пакета, что открывает доступ к принадлежащему этому контексту именам.

внутренний-вызов-глобальной-процедуры =

оператор-процедуры

{"OR" оператор-процедуры}

["ELSE" альтернатива] .

альтернатива = оператор .

**Примечание.** Семантика оператора внутреннего вызова описана в п. 6.2.

## 10. Объявление процедур и функций

объявление-процедуры =

заголовок-глобальной ";" блок |

заголовок-процедуры ";" блок |

заголовок-процедуры ";" директива .

заголовок-процедуры =

[ограничение-видимости]

"PROCEDURE" имя

["(" список-формальных-параметров ")"] .

директива =

"PASCAL" | "EXTERN" | "FORTRAN" |

"PROCESS" "(" имя-процесса ")" .

имя-процесса = полная-переменная | имя-константы .

заголовок-глобальной = внешний-заголовок

[ внутренний-заголовок ] .

внешний-заголовок = "GLOBAL" заголовок-процедуры .

внутренний-заголовок = "—>" [ имя ]

["(список-формальных-параметров)"] .

**Примечание.** Внешний заголовок глобальной процедуры задает интерфейс процедуры, доступный процессам-пользователям, т. е. ее внешнее имя, известное процессам-пользователям, и список параметров, используемый при удаленном вызове. Внутренний заголовок определяет интерфейс глобальной процедуры с ее процессом-владельцем. Таким образом, глобальная процедура имеет два существенно различных заголовка и соответственно два разных интерфейса. Директива PROCESS определяет имя процесса-владельца объявляемой удаленной глобальной процедуры. Имя процесса задается либо константой типа ALFA, либо переменной этого же типа.

объявление-функции =

заголовок-функции ";" блок |

заголовок-функции ";" директива .

заголовок-функции =

[ ограничение-видимости ]

"FUNCTION" имя

["("список-формальных-параметров")"]

":" тип .

**Примечание.** Возвращаемое значение функции определяется присваиванием его имени функции в теле блока функции. Для структурных функций допускается покомпонентное присваивание значения функции. В этом случае имя функции в теле ее используется точно так же, как если бы это было имя массива или записи,

список-формальных-параметров =

раздел-формальных-параметров

{";" раздел-формальных-параметров} .

раздел-формальных-параметров =

группа-параметров |

"VAR" группа-параметров |

заголовок-процедуры |

заголовок-функции .

группа-параметров = имя { "," имя } ":" тип .

**Примечание.** Имеются четыре вида формальных параметров: параметры-константы, параметры-переменные, параметры-процедуры и параметры-функции. Во внешнем заголовке глобальной процедуры и соответственно в объявлении ее в процессе-пользователе можно использовать только параметры-константы и параметры-процедуры. Фактическим параметром, соответствующим параметру-процедуре глобальной процедуры, может быть только имя глобальной же процедуры. Имена структурных функций не могут быть фактическими параметрами процедур и функций.

блок =

{ раздел-объявления }

[ условие-входа ]

составной-оператор .

условие-входа = "WHEN" выражение .

Примечание. Условие входа может задаваться только в теле глобальной процедуры.

раздел-объявления =

раздел-присоединения |

раздел-меток |

раздел-констант |

раздел-типов |

раздел-переменных |

раздел-инициализации |

объявление-процедуры |

объявление-функции .

раздел-меток =

"LABEL" метка { "," метка } "," .

раздел-констант =

"CONST" определение-константы ","  
{ определение-константы "," } .

раздел-типов =

"TYPE" определение-типа ","  
{ определение-типа "," } .

раздел-переменных =

"VAR" объявление-переменной ","  
{ объявление-переменной "," } .

раздел-инициализации =

"INITIALVALUES"  
инициализация-переменной ","  
{инициализация-переменной ","} .

ограничение-видимости =

"RESTRICTED" ["( список-видимости ")"] .

список-видимости = имя { "," имя } .

Примечание. Ограничение видимости без списка видимости делает недоступными в теле процедуры все имена, объявленные в объемлющих блоках, кроме предопределенных имен (стандартные процедуры и функции, предопределенные типы и константы). Список видимости делает доступными, кроме предопределенных, еще и указанные в нем имена (при условии, что эти имена были бы доступны, если бы ограничение видимости отсутствовало бы)

## 11. Стандартные процедуры и функции

### 11.1. Стандартные процедуры.

Описания стандартных процедур языка даются в виде их условных заголовков. Эти заголовки не всегда удовлетворяют правилам языка. Процедуры, соответствующие известным версиям языка Паскаль, подробно не описываются.

**PROCEDURE RESET(VAR F:FILE [ ; DDN:ALFA ] );**

Открытие файла для чтения. Необязательный параметр DDN идентифицирует набор данных. При реализации в ОС/ЕС — это имя оператора DD, описывающего требуемый набор данных.

**PROCEDURE REWRITE(VAR F:FILE [ ; DDN:ALFA ] );**

Открытие файла для записи.

**PROCEDURE EXTEND(VAR F:FILE [ ; DDN:ALFA ] );**

Открытие файла для добавления в него новых записей.

**PROCEDURE GET(VAR F:FILE);**

Чтение следующей компоненты файла в буфер файла.

**PROCEDURE PUT(VAR F:FILE);**

Запись новой компоненты в файл из буфера файла.

**PROCEDURE READ(VAR F:FILE; V1, ... , Vn : T);**

Чтение очередных компонент файла в указанные переменные.

**PROCEDURE WRITE(VAR F:FILE; V1, ... , Vn : T);**

Запись новых компонент файла из указанных переменных.

**CLOSE(VAR F:FILE);**

Закрытие файла.

**PROCEDURE NEW(VAR R:POINTER);**

Динамическое создание нового экземпляра переменной того типа, на который указывает ссылочный тип POINTER.

**PROCEDURE NEW(VAR R:POINTER; C1, ...<sup>м</sup>, Cm);**

Динамическое создание записи с вариантами, C1, ..., Cm — константы, определяющие значения дискриминантов записи.

**PROCEDURE DISPOSE(VAR R:POINTER);**

Динамическое освобождение памяти, занятой ранее размещенной переменной.

**PROCEDURE DISPOSE(VAR R:POINTER; C1, ... , Cm);**

Освобождение записи с вариантами.

**PROCEDURE MARK(VAR R:POINTER);**

Присваивание ссылочной переменной адреса конца области памяти, занятой динамическими переменными.



PROCEDURE RELEASE(R:POINTER);

Освобождение всей динамической памяти до границы, определенной значением переменной R.

PROCEDURE TIME(VAR S:ALFA);

Возвращает время дня в виде ЧЧ.ММ.СС.

PROCEDURE DATE(VAR S:ALFA);

Возвращает текущую дату в виде ГГ/ММ/ДД.

PROCEDURE MESSAGE(S:STRING);

Вывод сообщения S на консоль оператора ОС.

PROCEDURE HALT [ (S:STRING) ] ;

Аварийное прерывание выполнения программы с печатью сообщения S.

PROCEDURE SETCC(N:INTEGER);

Установка значения кода завершения программы.

PROCEDURE GETPARM(VAR S:STRING; VAR N:INTEGER);

Возвращает строку символов длиной N, переданную программе через поле PARM оператора языка управления заданиями ОС.

## 11.2. Стандартные функции.

FUNCTION ODD(X:INTEGER): BOOLEAN;

Возвращает TRUE, если X — нечетное число.

FUNCTION EOF(VAR F:FILE): BOOLEAN;

Возвращает TRUE, если достигнут конец файла F.

FUNCTION UNDEFINED(VAR X:T): BOOLEAN;

Возвращает TRUE, если значение переменной X не определено.

FUNCTION ABS(X:INTEGER): INTEGER;

FUNCTION ABS(X:REAL): REAL;

Абсолютное значение X.

FUNCTION SQR(X:INTEGER): INTEGER;

FUNCTION SQR(X:REAL): REAL;

$SQR(X) = X^{**}2$

FUNCTION SQRT(X:INTEGER): INTEGER;

FUNCTION SQRT(X:REAL): REAL;

Возвращает квадратный корень из X.

FUNCTION EXP(X:INTEGER): INTEGER;

FUNCTION EXP(X:REAL): REAL;

FUNCTION LN(X:INTEGER): INTEGER;

FUNCTION LN(X:REAL): REAL;

FUNCTION SIN(X : INTEGER) : INTEGER;

FUNCTION SIN(X : REAL) : REAL;

FUNCTION COS(X : INTEGER) : INTEGER;

FUNCTION COS(X : REAL) : REAL;

FUNCTION ARCTAN(X : INTEGER) : INTEGER;

FUNCTION ARCTAN(X : REAL) : REAL;

FUNCTION TRUNC(X : REAL) : INTEGER;

Возвращает значение X без дробной части.

FUNCTION ROUND(X : REAL) : INTEGER;

Ближайшее целое к X.

FUNCTION ORD(X : T) : INTEGER;

Если X любой простой тип кроме REAL, то значение — целое число, ассоциированное со значением X типа T; если T — ссылочный тип, то значение есть адрес, преобразованный в целое число.

FUNCTION CHR(X : INTEGER) : CHAR;

Возвращает символ, с кодом равным X.

FUNCTION SUCC(X : T) : T;

Возвращает значение следующее за значением X во множестве значений типа T.

FUNCTION PRED(X : T) : T;

Предыдущее значение типа T.

FUNCTION CLOCK : INTEGER;

Возвращает затраченное при выполнении данной программы время процессора.

FUNCTION CLOCKLEFT : INTEGER;

Возвращает время процессора, оставшееся до истечения указанного в задании лимита времени.

FUNCTION CARD(S : SET) : INTEGER;

Возвращает количество элементов множества S.

FUNCTION SIZESTACK : INTEGER;

Возвращает размер памяти, занятой стеком локальных переменных процедур.

FUNCTION SIZEHEAP : INTEGER;

Возвращает размер памяти, занятой динамически размещаемыми переменными.

FUNCTION SIZEFREE : INTEGER;

Возвращает размер свободной памяти.

**FUNCTION REQCOUNT(P : ALFA) : INTEGER;**

Возвращает количество запросов в очереди на вызов глобальной процедуры P. Может использоваться только в теле процесса-владельца глобальной процедуры.

**FUNCTION USER : ALFA;**

Возвращает имя процесса-пользователя, вызвавшего глобальную процедуру, в теле которой используется функция USER. Может использоваться только в теле глобальной процедуры.

### 11.3. Ф у н к ц и и з м е н е н и я т и п а .

Любое имя типа может быть использовано в качестве так называемой функции изменения типа. Применение такой функции по отношению к некоторому значению указывает компилятору, что это значение следует трактовать, как значение указанного типа. При этом никакого преобразования внутреннего представления значения не происходит. Ответственность за правильное применение функций преобразования типа лежит целиком на программисте.

## 12. Процедуры ввода-вывода для текстовых файлов

Процедуры ввода-вывода для текстовых файлов предусматривают преобразование данных из внутреннего представления в символьное при выводе или обратное преобразование при вводе.

**PROCEDURE READ( [ VAR F : TEXT ] V1, ..., Vn );**

Ввод из текстового файла значений типа INTEGER, REAL, CHAR, ALFA и PACKED ARRAY[1..N] OF CHAR.

**PROCEDURE READLN( [ VAR F : TEXT ] V1, ..., Vn );**

Эквивалентна процедуре READ, но после ввода пропускается остаток входной строки текстового файла.

**PROCEDURE WRITE( [ VAR F : TEXT ] P1, ..., Pn );**

Вывод символьного представления значений типов INTEGER, BOOLEAN, REAL, CHAR, ALFA, PACKED ARRAY[1..N] OF CHAR. P<sub>i</sub> может иметь вид V<sub>i</sub>:E, где E — формат вывода.

**PROCEDURE WRITELN( [ VAR F : TEXT ] P1, ..., Pn );**

После вывода производится смена строки на устройстве вывода.

**PROCEDURE PAGE [ (VAR F : TEXT) ];**

Смена страницы на печатающем устройстве.

**FUNCTION EOLN [ (VAR F : TEXT) ] : BOOLEAN;**

Значение функции равно TRUE, если достигнут конец строки файла F.

FUNCTION LINELENGTH(VAR F: TEXT): INTEGER;

Возвращает длину остатка текущей строки вводимого файла.

### 13. Пакеты

пакет =

[ограничение-видимости]  
[заголовок-пакета]  
{раздел-пакета}.

раздел-пакета =

[ "LOCAL" ] раздел-объявления |  
"PRIVATE" раздел-типов .

заголовок-пакета =

"PACKAGE" имя  
["(" список-формальных ")" ] ";," .

список-формальных =

группа-формальных  
{ ";," группа-формальных } .

группа-формальных = имя { ", " имя } ":" вид-формального .

вид-формального = "CONST" | "TYPE" | "INOUT" | "OUT" .

#### П р и м е ч а н и я.

1. Имя пакета, указываемое в заголовке, обычно совпадает со внешним именем пакета, под которым текст пакета хранится в библиотеке во внешней памяти. Однако это требование не обязательно. Это имя играет роль только в случае, если оно совпадает с именем одного из объявляемых в пакете объектов. Это случай пакета-функции.

2. Синтаксис пакета допускает *пустой* пакет, в котором есть только комментарий. Использование такого пакета не оказывает никакого влияния на компиляцию использующей его программы, за исключением того, что комментарии из пакета могут распечатываться в листинге компиляции.

3. Ограничение видимости пакета имеет тот же синтаксис, что и ограничение видимости процедуры. Оно действует только при вложении пакета, но не присоединении его,

раздел-присоединения =

"USE" спецификация-пакета  
{ ", " спецификация-пакета } ";,"

спецификация-пакета =

уточненное-имя |  
"CONTEXT" "(" имя-вложенного-контекста ")"  
[ имя-контекста "=" ] внешнее-имя-пакета  
[ параметры-пакета ] .

имя-вложенного-контекста = уточненное-имя .

внешнее-имя-пакета = имя .

имя-контекста = имя .

параметры-пакета =

"(" [ позиционные-параметры ]  
[ ключевые-параметры ] ")" .

позиционные-параметры =

позиционный-параметр

{ "," позиционный-параметр } [ "," ] .

позиционный-параметр =

константа | тип | имя |

уточненное-имя | пусто .

ключевые-параметры = { раздел-ключевых-параметров } .

раздел-ключевых-параметров =

раздел-констант |

раздел-типов .

**П р и м е ч а н и е.** Символ ";" после последнего раздела ключевых параметров не ставится, Семантика средств пакетирования описана в главе 3.

#### 14. Программа

программа = заголовок-программы блок "." .

заголовок-программы =

( "PROGRAM" | "PROCESS" ) имя

"(" файловая-переменная

{ "," файловая-переменная } ")" ";" .

**П р и м е ч а н и е.** Зарезервированное слово PROGRAM указывается в заголовке обычных последовательных программ; PROCESS указывает на то, что это процесс, выполняемый совместно с другими процессами. Средства удаленного вызова глобальных процедур можно использовать только в процессах. Использование их в последовательной программе вызовет ошибку при компиляции.

## СПИСОК ЛИТЕРАТУРЫ

1. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка.— М.: Финансы и статистика, 1982.
2. Форсайт Р. Паскаль для всех.— М.: Машиностроение, 1986.
3. Перминов О. Н. Язык программирования Паскаль.— М.: Радио и связь, 1983.
4. Liskov B. et al. CLU. Reference manual.— In: Lect. notes in comp. sci. № 114, 1981.—184 p.
5. ALPHARD: form and content.— M. Shaw, ed./Springer-Verlag, New York, 1981.
6. Вегнер П. Программирование на языке Ада.— М.: Мир, 1983.
7. Programming languages — Pascal. International standard.— ISO 7185—1983(E).
8. Данные в языках программирования/Под ред. Д. Б. Подшивалова.— М.: Мир, 1982.
9. Вирт Н. Программирование на языке Модула-2.— М.: Мир, 1987.
10. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.— М.: Мир, 1982.
11. Андерсон Р. Доказательство правильности программ.— М.: Мир, 1987.
12. Clark L., Koebler S. The UCSD Pascal handbook; A reference and guidebook for programmers.— Engl. Cliffs. Prentice Hall, 1982.
13. Вирт Н. Алгоритмы + Структуры данных = Программы.— М.: Мир, 1985.
14. Кнут Д. Искусство программирования для ЭВМ. Т. 1: Основные алгоритмы.— М.: Мир, 1976.
15. Калинин А. Сначала был только кубик // Наука и жизнь. — 1984.— № 9. — С. 97—103.
16. Элементы параллельного программирования.— Вальковский В. Л., Котов В. Е., Марчук А. Г., Миренков Н. И./Под ред. В. Е. Котова.— М.: Радио и связь, 1983.
17. Системы параллельной обработки/Под ред. Д. Ивенса.— М.: Мир, 1985.
18. Hansen P. B. The programming language Concurrent Pascal.— IEEE Trans. on Softw. Eng. № 1, 2 June 1975.— P. 199—207.
19. Дейкстра Э. Взаимодействие последовательных процессов // Языки программирования / Под ред. Ф. Женюи.— М.: Мир, 1972.

20. Дедков А. Ф., Щерс А. Л. Концепции взаимодействия процессов в языках программирования для сетей ЭВМ // Проблемы МСНТИ.— 1982.— № 2.— С. 47—53.
21. Andrews G. R., Schneider F. B. Concepts and notations for concurrent programming.— ACM computing surveys, V. 15, № 1, 1983.

### Список дополнительной литературы

1. Бар Б. Язык Ада в проектировании систем.— М.: Мир, 1988.
2. Вирт Н. Систематическое программирование. Введение.— М.: Мир, 1977.
3. Гласс Р. Руководство по надежному программированию.— М.: Финансы и статистика, 1982.
4. Грис Д. Наука программирования.— М.: Мир, 1984.
5. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов.— М.: Мир, 1981.
6. Дейкстра Э. Дисциплина программирования.— М.: Мир, 1978.
7. Замулин А. В., Скопин И. Н. Вычисления над типами // Программирование.— 1985, № 2.— С. 3—14.
8. Замулин А. В., Скопин И. Н. Конструкции языка программирования как типы данных // Прикладная информатика. Вып. 2(11), 1986.— С. 68—92.
9. Замулин А. В. Типы данных в языках программирования и базах данных.— Новосибирск: Наука, 1987.
10. Керниган Б., Плотджер Ф. Инструментальные средства программирования на языке Паскаль.— М.: Радио и связь, 1985.
11. Пайл Я. Ада — язык встроенных систем.— М.: Финансы и статистика, 1984.
12. Уэзерелл Ч. Этюды для программистов.— М.: Мир, 1982.
13. Хьюз Дж., Мичтом Дж. Структурный подход к программированию.— М.: Мир, 1980.
14. Harland D. M. Polymorphic programming languages: design and implementation.— 1984.

Научное издание

*ДЕДКОВ Анатолий Федорович*

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ В ЯЗЫКЕ АТ-ПАСКАЛЬ

«Библиотечка программиста», выпуск 58

Заведующий редакцией *А. С. Косов*

Редактор *О. Ю. Меркадер*

Художественный редактор *Т. Н. Кольченко*

Технический редактор *Л. В. Лихачева*

Корректор *И. Я. Кришталь*

ИБ № 32688

Сдано в набор 06.10.88. Подписано к печати 15.05.89. Т-10916.  
Формат 84×108/32. Бумага тип. № 2. Гарнитура литературная.  
Печать высокая. Усл. печ. л. 10,5. Усл. кр.-отт. 10,71. Уч.-изд. л.  
12,19. Тираж 33 000 экз. Заказ № 350. Цена ~~10~~ коп.

---

Ордена Трудового Красного Знамени издательство «Наука»  
Главная редакция физико-математической литературы  
117071 Москва В-71, Ленинский проспект, 15

---

Ордена Октябрьской Революции  
и ордена Трудового Красного Знамени  
МПО «Первая Образцовая типография»  
Государственного комитета СССР по делам издательств,  
полиграфии и книжной торговли. 113054 Москва, Валовая, 28

Отпечатано во 2-й типографии издательства «Наука».  
121099 Москва Г-99, Шубинский пер., 6. Зак. № 3019



80 коп.

581'60

Д-262