СОДЕРЖАНИЕ

Kathleen Jensen
Niklaus Wirth

Pascal User Manual and Report

Revised for the ISO Pascal Standard

Third Edition, Prepared by
Andrew B. Mickel
James F. Miner

Springer-Verlag
New York Berlin Heidelberg Tokyo

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

К.Йенсен, Н.Вирт

ПАСКАЛЬ
Руководство для пользователя

Перевод с английского Д. Б. Подшивалова

МОСКВА
"ФИНАНСЫ И СТАТИСТИКА"
1989

ББК 24.4.1
Й30

Йенсен К., Вирт Н.

Й30 Паскаль: руководство для пользователя/Пер. с англ. и предисл. Д. Б. Подшивалова. — М.: Финансы и статистика, 1989. — 255 с.: ил.

ISBN 5-279-00250-X.

Книга английских авторов содержит полное описание современной версии алгоритмического языка Паскаль, одного из самых популярных языков программирования. Новое издание расширено и переработано в соответствии с принятым стандартом ИСО. Книга может быть использована и как учебник для изучающих Паскаль, и как справочное руководство программиста.

Для специалистов в области информатики.

Й $\frac{2404010000 - 059}{010(01) - 89}$ 132 — 89 ББК 24.4.1

ISBN 3-540-96048-1 (англ.)

ISBN 5-279-00250-X (рус.)

ПРЕДИСЛОВИЕ К РУССКОМУ ИЗДАНИЮ

На протяжении ряда лет язык программирования Паскаль чаще других языков упоминается как в учебной, так и в научной литературе. Предлагаемая вниманию советского читателя книга представляет собой перевод уже третьего издания введения в программирование на Паскале, написанного первоначально создателем языка Н. Виртом совместно с К. Йенсен. Первая книга, вышедшая в узкоспециализированной, но хорошо известной серии «Lecture notes in computer science» в издательстве Шпрингер-Ферлаг, была затем выпущена вторым (массовым) изданием и теперь, после появления стандарта языка Паскаль, в переработанном виде выпускается третьим изданием, где (и это весьма существенно) приводится описание языка, эквивалентное описанию стандарта.

Созданный специально с педагогическими целями язык Паскаль оказался крайне удачным в силу того, что ему просто научиться. Кроме того, он стал основой обсуждения языков программирования вообще и своеобразным эталоном для сопоставлений. Благодаря концептуальной простоте Паскаля на его основе пытаются построить новые языки программирования, акцентируя внимание на тех или иных его особенностях. И происходит это в эпоху, когда почти в каждом свежем номере журнала по программированию можно найти если не описание, то ссылку на существование нового, доселе неизвестного языка. На наш взгляд, такую популярность Паскаля можно объяснить следующим.

Во-первых, язык проектировался (это неуклюжее слово никак не выражает творческий характер процесса «порождения» нового языка) с учетом простоты написания соответствующего транслятора. В результате создание такого транслятора почти не превышает по трудоемкости хорошую дипломную работу выпускника вуза. Небольшой объем трансляторов позволил достаточно подробно их описывать и хорошо документировать. Были разработаны специальные методики создания трансляторов с языка Паскаль, что привело к их широкому распространению.

Во-вторых, сам язык оказался очень простым. После «пудовых» описаний таких «монстров», как Алгол-68, ПЛ/1 или Ада, описание

Паскаля представляется неким откровением и напоминает старые добрые времена, когда, прочитав несколько страниц описания системы команд машины, можно было начинать писать программу. Причем (и это, пожалуй, самое существенное) выяснилось, что Паскаль позволяет писать программы весьма и весьма сложной природы, для которых популярные языки Фортран и Кобол оказались не совсем пригодными. Поэтому, в частности, Паскаль стал очень широко использоваться, например, на мини-машинах для программирования «встроенного» программного обеспечения уже как язык не «числённого», а системного программирования.

Это, так сказать, органически присущие языку особенности, способствовавшие его популярности, но существует некоторая более субъективная причина: хорошее описание. Книга состоит из двух достаточно автономных частей: «Описания языка», которое по аналогии с «Пересмотренным сообщением по Алголу-60» первоначально называлось «Сообщением», и «Руководства для пользователя» — учебника, причем не учебника по программированию, а учебника самого языка. В первых двух изданиях упомянутые части вместе образовывали то, что можно было бы назвать «классическим» учебником языка программирования. Четкие, ясные формулировки, почти полное отсутствие сложных формализмов делали книгу легко воспринимаемой как специалистом по языкам программирования, так и студентом, берущимся за составление своей первой программы. В прошедшие годы за рубежом появилось много учебников по программированию на Паскале, но книга так и оставалась непревзойденной. Появление стандарта Паскаля привело к необходимости появления третьего, весьма переработанного издания, перевод которого предлагается читателю. Остается надеяться, что оно поддержит популярность двух первых изданий.

Д. Б. Подшивалов

ВВЕДЕНИЕ

На протяжении почти десяти лет книга «Паскаль: руководство для пользователя и описание языка» служила стандартным учебником и справочником для программистов-практиков, желавших изучить язык Паскаль и пользоваться им. В 70-х годах популярность Паскаля превзошла все ожидания и он стал одним из наиболее значимых языков программирования, получивших распространение во всем мире. В этот момент коммерческое использование языка Паскаль в Соединенных Штатах часто обгоняло «академические интересы». Сегодня большинство университетов используют Паскаль для обучения программированию. Этот язык сейчас считается соперником языка ПЛ/1 или Алгола-60, даже Фортран начал изменяться, «осваивая» нововведения Паскаля*.

Работая в Pascal User's Group и *Pascal News***, мы были свидетелями распространения реализаций Паскаля на все современные вычислительные машины. В 1971 г. транслятор с Паскаля был всего лишь на одной машине. К 1974 г. число трансляторов выросло до десяти, а в 1979 г. их уже было более восьмидесяти. Язык Паскаль всегда присутствует и при работе на этих вездесущих потомках вычислительных систем: персональных ЭВМ и профессиональных рабочих станциях.

Проблемы, обсуждающиеся на симпозиуме по языку Паскаль в 1977 г. (Соутхемптон) [10], привели к первой организованной попытке написать некоторый официально санкционированный международный стандарт Паскаля. Участники семинара стремились составить общий список вопросов, которые, естественно, возникали у людей, пытавшихся реализовать трансляторы с Паскаля, основываясь на определениях, приводившихся в книге «Паскаль: руководство для пользователя и описание языка». Эти усилия в конце концов закончились созданием международного стандарта Паскаля (ISO Pascal Standard) [11] — документа, официально

---

* Это очень смелое утверждение. — *Примеч. пер.*

** Первое — неформальное объединение пользователей Паскаля, а второе — печатное издание упомянутого объединения. — *Примеч. пер.*

определяющего язык Паскаль и приведшего к необходимости пересмотра нашей книги.

На нас была возложена задача модификации «Руководства для пользователя и описания языка» применительно к этому стандарту. Наша книга не стремится заменить стандартное описание. Мы надеемся, что нам удалось во многом сохранить те первоначальные удобочитаемость и элегантность, которые, как нам кажется, существенно отличают ее от стандарта. Для определения синтаксических конструкций мы использовали употребляемую Н. Виртом систему EBNF (РБНФ — Расширенные Бэкуса — Наура Формы*). Кроме того, мы несколько улучшили стиль программ, приведенных в «Руководстве для пользователя». Для удобства читателей, уже знакомых с предыдущими изданиями книги, в приложение 5 включены все изменения, продиктованные стандартом.

И наконец, следует отметить, что «Паскалем» язык назвали в честь французского математика, гуманиста и религиозного фанатика — Блеза Паскаля, человека, построившего первую простую вычислительную машину**. Мы хотим поблагодарить Роберта Миньо и Никласа Вирта за их поддержку проекта пересмотра книги. Очень много внимания уделил нам Генри Ледгард, его советы были весьма ценными и полезными. Элиза Оранж добросовестно следила за порядком. Мы также благодарим В. В. Портера за оформление, а Линду Стржеговски за набор нашей книги.

Миннеаполис, США
Ноябрь, 1984

Энди Майкель
Джим Майнер

---

* При переводе эта система была несколько «руссифицирована»: в новом издании книги синтаксические конструкции идентифицируются «длинным» словом, состоящим из нескольких обычных. Последние друг от друга ничем не отделяются, но на «германский» манер каждое слово начинается с прописной буквы. Мы же будем разделять их пробелами, причем первое слово такой комбинации будет начинаться с прописной буквы, а остальные — со строчной. Надеемся, что такая нотация с присущей русскому языку системой управления с помощью падежных окончаний позволяет надежно выделять синтаксические конструкции. — *Примеч. пер.*

** Может быть, это замечание приведет к тому, что некоторые авторы и программисты перестанут называть язык Паскаль «паскалем». — *Примеч. пер.*

ПРЕДИСЛОВИЕ

Предварительное описание языка программирования Паскаль было опубликовано в 1968 г. Это был язык, по духу продолжавший линию языков Алгол-60 и Алгол-W. Затем, после периода интенсивного развития языка, в 1970 г. заработал первый транслятор, а годом позже последовали соответствующие публикации [1, 8]. Растущий интерес к созданию трансляторов на других машинах привел к распространению языка, и после двух лет его использования потребовалось ввести в язык небольшие изменения. Поэтому в 1973 г. было опубликовано «Пересмотренное сообщение», где язык был уже определен в терминах множества символов ИСО.

Эта книга состоит из двух частей: руководства для пользователя и пересмотренного сообщения. Руководство предназначается для тех читателей, которые уже знакомы с программированием для вычислительных машин и хотят получить представление о языке Паскаль. Поэтому оно написано как учебник, в нем много примеров, демонстрирующих те или иные особенности языка. В приложениях же приводятся обзорные таблицы и синтаксические определения. Включенное в книгу «Описание языка» должно служить в качестве точного, исчерпывающего справочника как для программистов, так и для специалистов, занимающихся реализацией языка. Оно описывает стандартный Паскаль — основу различных реализаций языка.

Линейная структура, присущая любой книге, не очень подходит для того, чттобы дать представление о языке программирования. Однако если вы используете эту книгу как учебник, то мы все же рекомендуем следовать именно выбранной организации материала, уделяя при этом особое внимание примерам программ, и только после этого уже перечитывать те разделы, которые вызвали затруднение. В частности, если возникают вопросы, связанные с соглашениями о вводе и выводе, то можно справиться о них в гл. 12.

В гл. 1—12 руководства и в описании рассматривается стандартный Паскаль. Любой разработчик, реализующий язык, должен прежде всего стремиться реализовать стандартный Паскаль — это основное требование к его системе. В то же время,

если программист хочет, чтобы его программы можно было переносить с одной машины на другую, он должен пользоваться только теми особенностями языка, которые включены в стандартный Паскаль. Конечно, отдельные разработчики могут включать в язык дополнительные возможности, но они должны явно отмечаться как расширения.

Многие люди способствовали созданию этой книги, и нам особенно хочется поблагодарить сотрудников Института информатики (ETH) в Цюрихе Джона Лармота, Раду Шильд, Оливера Лекарме, Пьера Дежардена за их критику, предложения и добрые пожелания. Реализация Паскаля, сделавшая возможным и полезным появление нашей книги, выполнена Урсом Амманом, которому помогал Гельмут Сандмайр.

ETH, Цюрих
Швейцария
Ноябрь, 1974

Кэтлин Йенсен
Никлас Вирт

РУКОВОДСТВО ДЛЯ ПОЛЬЗОВАТЕЛЯ

ВСТУПЛЕНИЕ

1. ОБЗОР ПРОГРАММЫ НА ПАСКАЛЕ

В большей части последующего текста предполагается, что читатель немного знаком с терминологией программирования и «чувствует» структуру программы. Цель настоящего вступления — напомнить эти интуитивные представления.

Алгоритм, или *программа,* для вычислительной машины состоит из двух важных разделов (частей): описания действий, которые необходимо выполнить, и описания данных, с которыми оперируют упомянутые действия. Действия описываются с помощью того, что называется *операторами,* а данные — с помощью *описаний* и *определений.*

Программа делится на *заголовок* и «тело» программы, называемое *блоком.* В заголовке программе дается имя и перечисляются ее параметры. Обычно это — переменные (файлы) и они представляют собой аргументы и результаты вычислений. Блок состоит из шести разделов, причем любой из них, кроме последнего, может быть пустым. В определении блока разделы должны следовать в порядке, указанном в следующем определении:

Блок = *Раздел описания меток*
Раздел определения констант
Раздел определения типов
Раздел описания переменных
Раздел описания процедур и функций
Раздел операторов.

Пример программы:

```
program Inflation(Output);

  { В предположении, что годовая скорость
    инфляции — 7%, 8% и 10%, определяется
    коэффициент девальвации таких валют,
    как франк, доллар, фунт стерлингов, марка,
    рубль, иена или гульден за 1, 2, . . . , n лет }
```

```
const
   MaxYears = 10;

var
   Year: 0..MaxYears;
   Factor1, Factor2, Factor3: Real;

begin
   Year := 0;
   Factor1 := 1.0;  Factor2 := 1.0;  Factor3 := 1.0;
   Writeln(' Year     7%     8%    10%');  Writeln;
   repeat
      Year := Year + 1;
      Factor1 := Factor1 * 1.07;
      Factor2 := Factor2 * 1.08;
      Factor3 := Factor3 * 1.10;
      Writeln(Year :5, Factor1 :7:3, Factor2 :7:3, Factor3 :7:3)
   until Year = MaxYears
end .
```

Дает в качестве результатов:

```
 Year     7%     8%    10%

    1  1.070  1.080  1.100
    2  1.145  1.166  1.210
    3  1.225  1.260  1.331
    4  1.311  1.360  1.464
    5  1.403  1.469  1.611
    6  1.501  1.587  1.772
    7  1.606  1.714  1.949
    8  1.718  1.851  2.144
    9  1.838  1.999  2.358
   10  1.967  2.159  2.594
```

В первом разделе перечисляются все метки, определяемые в данном блоке. Во втором — определяются синонимы для констант, т. е. вводятся «имена констант», которые можно позже использовать вместо самих констант. Третий раздел содержит определения типов, четвертый — переменных. В пятом разделе приво-

дятся определения «автономных» частей программы (т. е. процедур и функций). Раздел операторов задает сами действия, которые необходимо выполнить.

2. СИНТАКСИЧЕСКИЕ ДИАГРАММЫ

Предыдущее обсуждение строения программы можно дополнить графическими *синтаксическими диаграммами*. Если начать с диаграммы для понятия (конструкции) *Программа* (рис. 1), то

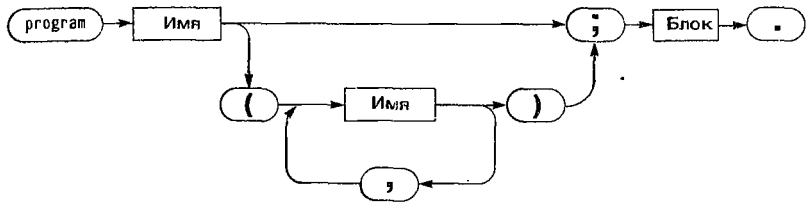


Рис. 1. Синтаксическая диаграмма для *Программы*

путь по диаграммам определит синтаксически корректную программу. Каждый прямоугольник содержит имя, указывающее на диаграмму, используемую для определения соответствующего «значения». Терминальные символы (т. е. символы, из которых состоит написанная на Паскале программа) помещаются в «кружки» или «овалы». (В приложении 4 приводится все множество диаграмм для Паскаля.)

3. РБНФ

Синтаксис языка можно описать с помощью еще одного метода: *Расширенных Бэкуса — Наура Форм* (РБНФ), где синтаксические конструкции обозначаются английскими (или русскими) словами и буквальными последовательностями символов. Считается, что упомянутые слова отражают природу или смысл конструкции, в то время как буквальные последовательности обозначают именно те символы, которые используются в нашем языке. Буквальные последовательности заключаются в кавычки.

Если несколько конструкций или буквальных последовательностей заключены в метасимволы { и }, то это означает, что они встречаются нуль или более раз. Альтернативы разделяются с помощью

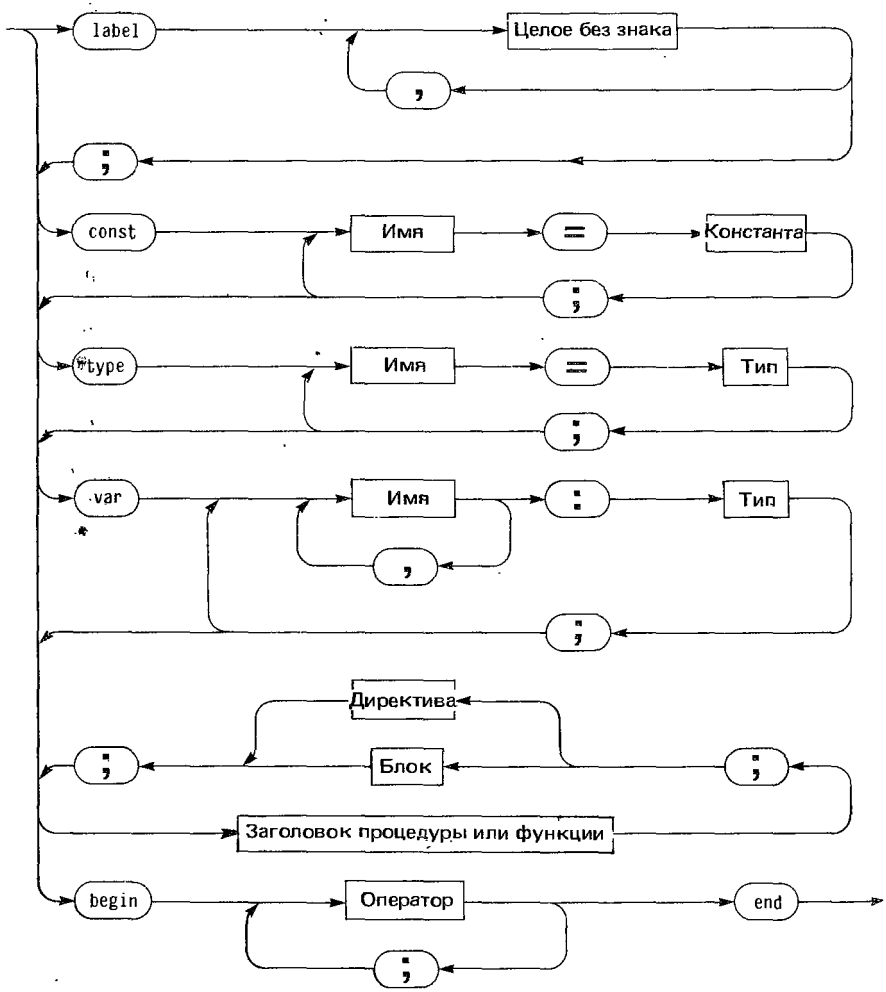


Рис. 2. Синтаксическая диаграмма для *Блока*

метасимвола |. Для простого группирования употребляются круглые скобки (и), а метасимволы [и] означают, что заключенные в них конструкции и буквальные последовательности могут встречаться, а могут и не встречаться. (Полное объяснение РБНФ и сами РБНФ для Паскаля приводятся в приложении 4.) Например,

конструкция *Программа,* приведенная на рис. 1, определяется с помощью таких РБНФ формул, называемых *порождающими правилами.*

Программа = *Заголовок программы*";" *Блок* ".".
Заголовок программы = "program" *Имя* ["("*Список имен* ")"].
Список имен = *Имя* {"," *Имя*}.

4. ОБЛАСТЬ ДЕЙСТВИЯ

Описание каждой процедуры и функции по структуре похоже на программу, т. е. оно состоит из заголовка и блока. Следовательно, описания процедур и функций могут вкладываться внутрь других процедур и функций. Описания меток, синонимов констант, типов, переменных и процедур, а также функций являются локальными по отношению к процедуре или функции, в которой они описаны. Это означает, что соответствующие имена имеют смысл только в тексте программы, составляющем соответствующий блок. Такой фрагмент текста называется *областью действия* этих имен. Поскольку блоки могут быть вложены один в другой, то вложенными могут быть и области действия. Объекты, описываемые в главной программе, т. е. не локализованные в какой-либо процедуре или функции, называются *глобальными* и доступны в любом месте программы.

Так как блоки с помощью описаний процедур и функций можно вкладывать один в другой, то каждому из них можно приписать некоторый уровень вложенности. Если самый внешний, определяемый программой блок, т. е. главная программа, относится к уровню 0, то блок, определенный внутри этого блока, следует отнести к уровню 1 и т. д., в общем случае блок, определенный на уровне i, относится к уровню $(i + 1)$. Пример на рис. 3 иллюстрирует структуру вложенных блоков. Такой структуре блоков соответствует следующая схема программы:

```
program M;
   procedure P;
      procedure A;
         procedure B;
         begin
         end { B };
      begin
      end { A };
   begin
   end { P };
```

```
    procedure Q;
       procedure R;
       begin
       end { R };
       procedure S;
       begin
       end { S };
    begin
    end { Q };
begin
end { M }.
```

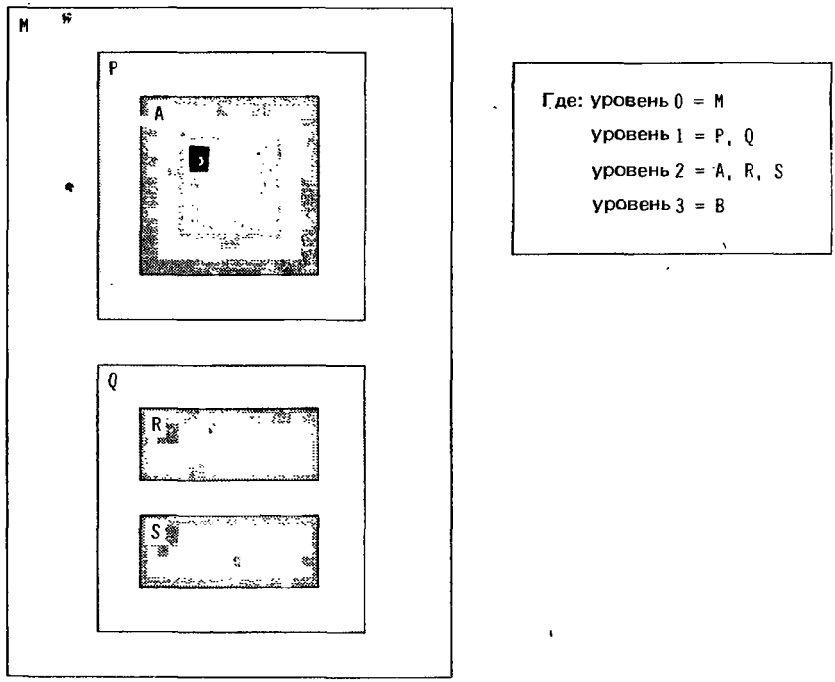


Рис. 3. Структура блоков

Область действия или диапазон доступности имени *X* представляет собою весь блок, в котором определяется *X*, включая и блоки, определенные в блоке, где определен *X*. (Обратите внимание, что

в нашем примере все имена должны быть различными. В разд. 3.6 рассматривается случай, когда имена не обязательно различны.)

блок	можно обращаться к объектам из блоков
M	M
P	P, M
A	A, P, M
B	B, A, P, M
Q	Q, M
R	R, Q, M
S	S, Q, M

5. РАЗНОЕ

Для программистов, знакомых с языками Алгол, ПЛ/1 или Фортран, можно кратко охарактеризовать язык Паскаль, перечислив его характерные особенности.

1. Обязательное описание переменных.

2. Служебные слова (например, begin, end, repeat), «резервируются» и не могут использоваться в качестве имен.

3. Точка с запятой (;) считается разделителем операторов.

4. Существуют стандартные типы для целых и вещественных чисел, логических значений и символов (тех, которые можно напечатать). Основные возможности построения данных со сложной структурой обеспечивают создание массивов, записей (в Коболе и ПЛ/1 им соответствуют «структуры»), множеств и (последовательных) файлов. Такие структуры могут комбинироваться и вкладываться одна в другую, образуя массивы множеств, файлы записей и т. д. Данные могут размещаться динамически и затем к ним можно обращаться через ссылки, позволяющие вести самые общие работы со списками. Существует возможность описывать новые, основные типы данных с «символическими» константами.

5. Данные типа множество обеспечивают те же возможности, что и объекты типа «строка разрядов» в ПЛ/1.

6. Массивы могут быть произвольной размерности, границы тоже произвольные. Однако они должны задаваться константами (это означает, что динамических массивов нет).

7. Как и в Алголе, Фортране, ПЛ/1 в языке Паскаль есть оператор перехода. Метки представляют собой целые числа и должны быть описаны.

8. Составной оператор такой же, как в Алголе, и соответствует оператору DO (группе) в ПЛ/1.

9. Такая возможность, как оператор переключателя в Алголе или вычисляемый переход в Фортране, в Паскале обеспечивается оператором варианта.

10. В операторе цикла с параметром, соответствующим циклу DO в Фортране, можно пользоваться либо шагом 1 (to), либо шагом — 1 (downto) и только. Цикл выполняется до тех пор, пока значение параметра (управляемой переменной) находится внутри указанных пределов. Следовательно, тело цикла может вообще ни разу не выполняться.

11. В языке нет ни условных выражений, ни кратных присваиваний.

12. Процедуры и функции допускают рекурсивное обращение.

13. У переменных отсутствует атрибут «own» (собственная), имеющийся в Алголе.

14. Параметры передаются либо «по значению», либо «по ссылке». Передачи «по имени» нет.

15. Структура блоков отличается от структуры блоков в Алголе и ПЛ/1, поскольку нет «анонимных» блоков: каждый блок имеет имя и тем самым превращается в процедуру.

16. Все объекты (константы, переменные и т. д.) должны быть описаны, *прежде* чем на них появятся ссылки (упоминания). Однако из этого правила есть два исключения:

а) для имени типа в описании ссылочного типа (гл. 10);

б) для имени процедуры или функции при опережающем описании (разд. 11.3).

При первом знакомстве с языком Паскаль многие программисты начинают «причитать» по поводу отсутствия в нем «удобных» конструкций. Упоминают и операции возведения в степень, и конкатенацию строк, и динамические массивы, и арифметические операции над логическими значениями, и автоматическое преобразование типов, и описание исключительных ситуаций. Но все это исключено из языка не по какой-либо оплошности, а в результате намеренных сокращений. Некоторые из таких конструкций стали бы «приглашениями» к неэффективным программным решениям, а некоторые, как нам казалось, должны были вступить в противоречие со стремлением к ясности, надежности и хорошему стилю программирования. Кроме всего прочего, строгий отбор из огромного множества имеющихся вариантов нужно проводить и для того, чтобы транслятор оставался относительно компактным и эффективным. Он должен быть экономным и эффективным как для тех пользователей, которые пишут только небольшие «программки» и используют лишь некоторые из конструкций языка, так и для тех, кто составляет большие программы и стремится пользоваться всеми возможностями языка.

1

НОТАЦИЯ: ЛЕКСЕМЫ И РАЗДЕЛИТЕЛИ

Программа на Паскале состоит из лексем и символов-разделителей. В лексемы Паскаля входят *специальные символы, символы-слова,* имена, числа, строки символов, метки и директивы. В следующем разделе речь пойдет о *символах-разделителях.*

1.1. РАЗДЕЛИТЕЛИ

Символами-разделителями считаются пробелы, концы строк (разделители строк) и примечания. Внутри лексем Паскаля ни разделители, ни их части встречаться не могут. Между двумя следующими друг за другом именами, символами-словами или числами должен быть по крайней мере один разделитель.

Примечание начинается с символа { или (* (но не внутри строки символов) и заканчивается } или *). Само примечание может содержать любые символы, включая концы строк, за исключением } или *). Любое примечание может быть заменено на пробел, смысл текста программы при этом не изменяется.

Вы можете улучшать внешний вид (удобочитаемость) программы на Паскале, включая в нее пробелы, концы строк (пустые строки) и примечания.

1.2. СПЕЦИАЛЬНЫЕ СИМВОЛЫ И СИМВОЛЫ-СЛОВА

Ниже приводится список специальных символов и символов-слов, употребляемых при написании программ на Паскале. Обращаем ваше внимание, что специальные символы, состоящие из двух символов, не допускают «вклинивания» в них разделителей. Вот эти специальные символы:

```
+   -   *   /
    '       ;
=   <>  <   <=  >   >=
:=  ..  ↑
(   )   [   ]
```

Существуют и альтернативные написания:

(. для [

.) для]

@ или ^ для ↑

Символы-слова (или зарезервированные слова) в рукописных программах обычно подчеркиваются, чтобы их легче было воспринимать как единые символы с фиксированным смыслом. Использовать эти слова в каком-либо другом значении кроме того, которое зафиксировано в определении Паскаля нельзя; в частности, эти слова не употребляются в качестве имен. Слова записываются как последовательности прописных и строчных букв без каких-либо «невидимых» символов. Вот эти слова-символы:

and	end	nil	set
array	file	not	then
begin	for	of	to
case	function	or	type
const	goto	packed	until
div	if	procedure	var
do	in	program	while
downto	label	record	with
else	mod	repeat	

1.3. ИМЕНА

Имена применяются для обозначения констант, типов, границ, переменных, процедур и функций. Они должны начинаться с бук-

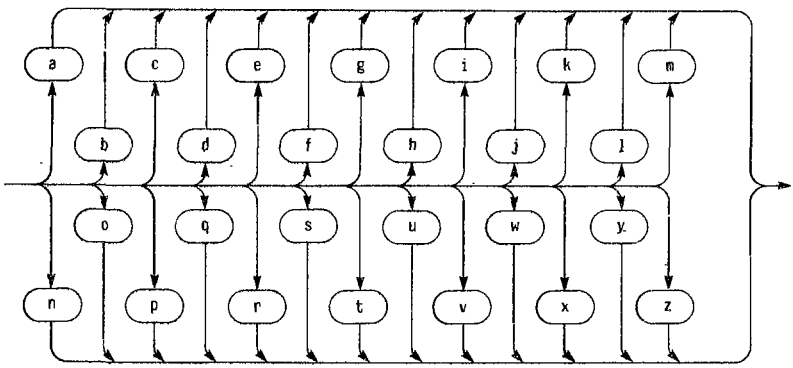


Рис. 1.1. Синтаксическая диаграмма для *Буквы*

вы, за которой могут следовать в любой комбинации любое число букв и цифр. Воспринимаются и имеют смысл все символы имени. Соответствующие строчные и прописные буквы считаются эквивалентными.

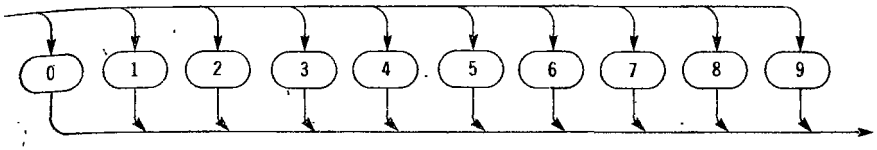


Рис. 1.2. Синтаксическая диаграмма для *Цифры*

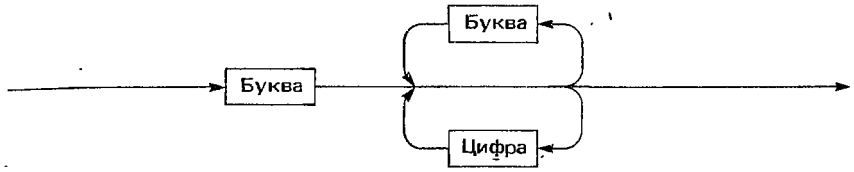


Рис. 1.3. Синтаксическая диаграмма для *Имени*

Примеры имен:

```
PhoneList   Root3   Pi   h4g   X
ThisIsAVeryLongButNeverTheLessValidIdentifier
ThisIsAVeryLongButDifferentIdentifierThanTheOneAbove
```

Имя LetterAndDigits эквивалентно letteranddigits.
А такие имена не допускаются:

```
3rd array level.4 Root-3 Tenth__Planet
```

Некоторые имена, называемые *предописанными именами,* предусмотрены заранее (например, sin, cos). В отличие от слов-символов (например, array), мы не накладываем каких-либо ограничений на их описание, поэтому можно переопределить любое предописанное имя; их можно считать описанными в некотором гипотетическом блоке, внутри которого находится блок программы. В приложении 3 приводится таблица со всеми предописанными именами Паскаля.

1.4. ЧИСЛА

Для *чисел*, обозначающих целые или вещественные значения, используется десятичная нотация. Перед любым числом может стоять знак (+ или —), однако в конструкции *число без знака* знак ставить нельзя. В числе не допускается никаких запятых. Вещественные числа записываются с десятичной точкой или с масштабным множителем (порядком), можно и с тем, и с другим. Буква E (или e), за которой следует порядок, читается как «умножить на десять в степени». Обратите внимание, что если вещественное число содержит десятичную точку, то перед нею и после нее должно быть по крайней мере по одной цифре.

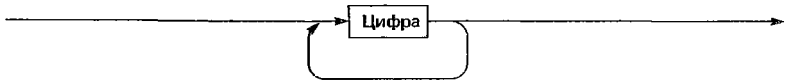


Рис. 1.4. Синтаксическая диаграмма для *Целого без знака*

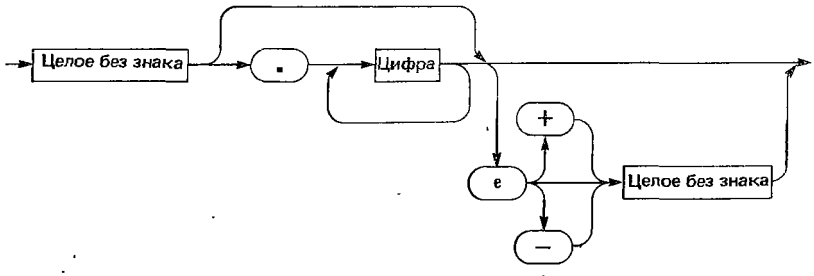


Рис. 1.5. Синтаксическая диаграмма для *Числа без знака*

Примеры чисел без знака:
3 03 6272844 0.6 5E—8 49.22E+08 1E10

Неправильно написанные числа:
3,487,159 XII .6 E10 5.E—16 five 3.487.159

1.5. СТРОКИ СИМВОЛОВ

Заключенные в апострофы (одиночные кавычки) последовательности символов называются *строками*. Если нужно включить в строку сам апостроф, то он записывается дважды.

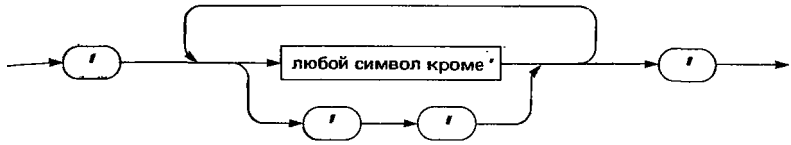


Рис. 1.6. Синтаксическая диаграмма для *Строки символов*

Примеры строк:

```
'a'    ';'    '3'    'begin'    'don''t'
'   This string has 33 characters.'
```

1.6. МЕТКИ

Метки представляют собой целые без знака и употребляются для маркировки операторов Паскаля. Их «значения» должны лежать в диапазоне от 0 до 9999.

Примеры меток:
13 00100 9999

1.7. ДИРЕКТИВЫ

Директивы — это имена, подставляемые вместо блоков процедур и функций. Синтаксис директив тот же, что и синтаксис имени (см. гл. 11).

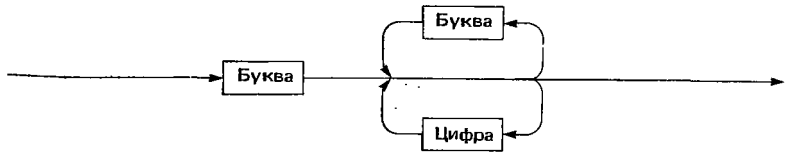


Рис. 1.7. Синтаксическая диаграмма для *Директивы*

2

КОНЦЕПЦИЯ ДАННЫХ: ПРОСТЫЕ ТИПЫ ДАННЫХ

Данные — это общее понятие для всего того, с чем оперирует вычислительная машина. В аппаратуре все данные представляются как последовательности двоичных цифр (разрядов), такими же мыслятся данные и при программировании на уровне машинных команд. Языки высокого уровня позволяют абстрагироваться от деталей представления, главным образом за счет введения концепции *типа данных.*

Любой тип данных определяет множество значений, которые может принимать та или иная переменная, и те операции, которые можно к ним применять. С каждой встречающейся в программе переменной должен быть сопоставлен один и только один тип. Хотя в Паскале типы данных крайне изощренные, тем не менее каждый из них должен в конце концов строиться из элементарных, простых данных.

В Паскале предусмотрены и возможности порождения типов для групп данных (речь идет о составных и ссылочных типах). Такие типы обсуждаются в гл. 6—10.

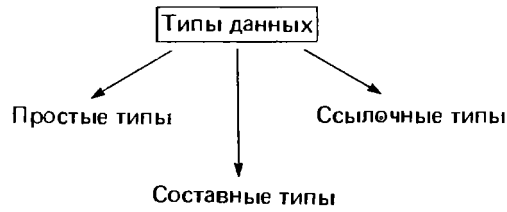


Рис. 2.1. Схема типов данных

В Паскале существуют простые типы двух видов: ординальные* типы и вещественный тип. Ординальный тип либо определяется

* В оригинале стоит слово «ordinal»; его точный перевод «порядковые». Ранее в аналогичной ситуации в книге употреблялся «scalar» — «скалярные». Стремясь к адекватному переводу, мы вводим слово «ординальные». — *Примеч. пер.*

программистом (в этом случае его называют перечисляемым типом или диапазоном), либо обозначается именем одного из трех предописанных ординальных типов — Boolean, Integer или Char. Вещественный тип обозначается именем предописанного типа Real.

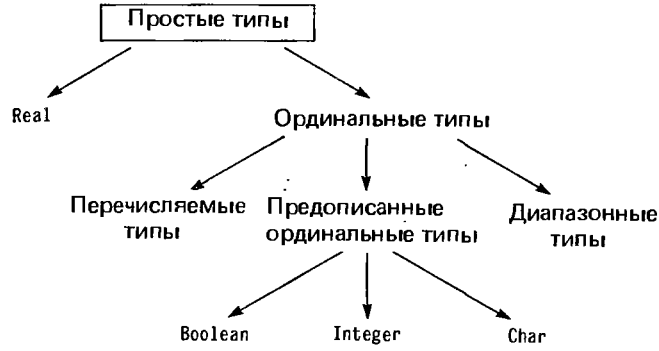


Рис. 2.2. Схема простых типов данных

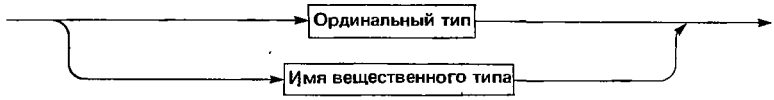


Рис. 2.3. Синтаксическая диаграмма для *Простого типа*

Перечисляемый тип характеризуется множеством входящих в него различных значений, среди которых определен линейный порядок. Сами значения обозначаются в определении этого типа именами. Диапазонный (ограниченный) тип задается с помощью минимального и максимального значений, относящихся к предварительно описанному ординальному типу. Так порождается новый ординальный тип. Перечисляемые и диапазонные типы рассматриваются в гл. 5.

2.1. ОРДИНАЛЬНЫЕ ТИПЫ ДАННЫХ

Ординальный тип данных описывает конечное и упорядоченное множество значений. Эти значения отображаются на последовательность *порядковых номеров* 0, 1, 2, ...; исключение делается

лишь для целых ординальных чисел, которые отображаются сами на себя. Каждый ординальный тип имеет минимальное и максимальное значение. Для всех значений кроме минимального существует *предшествующее* значение, а для всех значений кроме максимального — *последующее.*

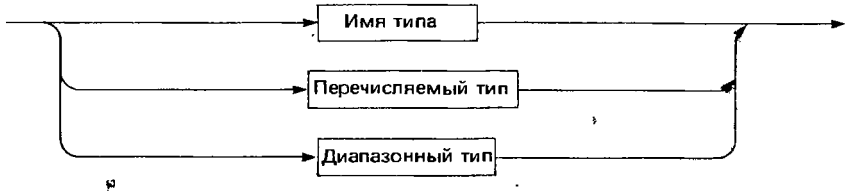


Рис. 2.4. Синтаксическая диаграмма для *Ординального типа*

Предописанные функции succ, pred и ord воспринимают аргументы любого из ординальных типов:

succ(X)	дает следующее за X ординальное значение
pred(X)	дает предшествующее X ординальное значение
ord(X)	дает ординальный номер для X

Для всех ординальных типов существуют операции отношения =, <>, <, <=, >= и >, причем предполагается, что оба операнда одного и того же типа. Отношение определяется с помощью значений ординальных номеров, присущих операндам.

2.2. ЛОГИЧЕСКИЙ ТИП (BOOLEAN)

Логическое значение — одно из двух истиностных значений, обозначаемых предопределенными именами false и true.

Существуют следующие логические операции, дающие логическое значение при применении их к логическим операндам (в приложении 2 приводятся все операции):

and	логическая конъюнкция
or	логическая дизъюнкция
not	логическое отрицание

Любая из операций отношения (=, <>, <=, <,>,>=, in) поставляет логический результат. Операция «<>» обозначает неравенство. Кроме того, сам логический тип определен так, что false<true. Следовательно, любую из 16 логических операций

можно определить с помощью приведенных выше логических операций и операций отношения. Если, например, P и Q — логические значения, то мы можем выразить:

импликацию как P <= Q;
эквивалентность как P = Q;
исключающее или как P <> Q.

Существуют и предописанные логические функции (т. е. функции, дающие логический результат):

odd(I)	true, если целое I — нечетное и результат false, если I — четное
eoln(F)	проверка на конец строки (объяснение в гл. 9)
eof(F)	проверка на конец файла (объяснение в гл. 9)

(В приложении 1 приводятся все предописанные функции.)

2.3. ЦЕЛЫЙ ТИП (INTEGER)

Значениями типа Integer являются элементы определяемого при реализации подмножества целых чисел.

При приложении к целым операндам следующие арифметические операции дают целые значения:

*	умножение
div	деление с «отсечением» (т. е. значение не округляется);
mod	остаток пусть Remainder = A — (A div B) * B; если Remainder < 0, то A mod B = Remainder + B; иначе A mod B = Remainder;
+	сложение
—	вычитание

Существует определяемая при реализации предописанная константа с именем MaxInt, дающая самое большое целое значение, допустимое для всех целых операций. Если A и B — целые выражения то гарантируется, что операция

A op B

будет выполнена верно при

```
abs(A op B)  <=  MaxInt,
abs(A)       <=  MaxInt, and
abs(B)       <=  MaxInt
```

Целый результат дают и четыре важные предописанные функции:

abs(I)	абсолютное значение целого значения I
sqr(I)	целое значение I, возведенное в квадрат при условии, что I <=MaxInt div I

trunc(R)	R — вещественное значение, результат — его целая часть. (Дробная часть отбрасывается. Следовательно, trunc (3.7) = = 3, trunc(—3.7) = —3
round(R)	R — вещественное значение, результат — округленное целое. round(R) для $R > 0$ означает trunc(R + 0.5), а для $R < 0$ — trunc(R — 0.5)

Если I — целое значение, то

succ(I)	дает «следующее» целое значение (I + 1)
pred(I)	дает «предыдущее» целое значение (I — 1).

2.4. СИМВОЛЬНЫЙ ТИП (CHAR)

Значениями типа Char являются элементы конечного и упорядоченного множества символов. В каждой вычислительной системе такое множество обязательно есть — оно необходимо для связи с системой. Символы этого множества есть и на вводных, и на выводных устройствах. К сожалению, не существует одного стандартного множества символов, поэтому и его элементы, и их упорядоченность обязательно определяются при реализации (см. приложение 6).

Значения такого типа обозначаются одним символом, заключенным в одиночные кавычки (апострофы).

Примеры:

'*' 'G' '3' '''' 'X'.

(Если нужен сам апостроф, то он пишется дважды.) Возможно, что некоторые из символьных значений не имеют такого «константного» представления.

Мы полагаем, что вне зависимости от реализации для типа Char справедливы следующие минимальные допущения:

1. Десятичные цифры от '0' до '9' упорядочены в соответствии с их числовыми значениями и следуют одна за другой (например, succ ('5') = '6').

2. Могут существовать прописные буквы от 'A' до 'Z'; если это так, то они упорядочены в алфавитном порядке, но не обязательно следуют одна за другой (например, 'A' < 'B').

3. Могут существовать строчные буквы от 'a' до 'z'; если это так, то они упорядочены в алфавитном порядке, но не обязательно следуют одна за другой (например, 'a' < 'b').

Для отображения заданного множества символов на порядковые номера и обратно существуют две предописанные функции ord и chr; будем называть их *функциями отображения* (transfer):

ord(C)	дает порядковый номер символа C в упомянутом упорядоченном множестве символов
chr(I)	дает символ с порядковым номером I

Очевидно, что функции ord и chr обратные по отношению друг к другу, т. е.

chr(ord(C)) = C и ord(chr(I)) = I

Более того, упорядоченность заданного множества определяется так, что C1 < C2, если и только если ord(C1) < ord(C2). Такое определение можно распространить и на любую из операций отношения: =, <>, <, <=, >=, >. Если через R обозначить одно из этих отношений, то C1 R C2, если и только если ord(C1) R ord(C2).

Для аргументов типа Char предописанные функции pred и succ могут быть определены таким образом:

```
pred(C) = chr(ord(C)-1)
succ(C) = chr(ord(C)+1)
```

Замечание. Предшествующий данному либо следующий за ним символ зависит от указанного множества символов, поэтому оба этих соотношения справедливы только в том случае, когда предшествующий или следующий символ существует.

2.5. ВЕЩЕСТВЕННЫЙ ТИП (REAL)

Значениями *вещественного* типа являются элементы определяемого реализацией подмножества вещественных чисел.

Все операции над величинами вещественного типа — приближенные, их точность определяется реализацией (машиной), с которой вы имеете дело. Вещественный тип относится к простому типу, это не ординальный тип. У вещественных значений нет ординального номера и для любого из них не существует предшествующего и следующего значений.

При условии, что хотя бы один из операндов — вещественного типа (другой может быть и целым), следующие операции дают вещественный результат:

*	умножение
/	деление (оба операнда могут быть целыми, но результат всегда вещественный)
+	сложение
—	вычитание

Существуют предописанные функции, дающие вещественный результат при вещественном аргументе:

abs(R)	абсолютное значение R
sqr(R)	R в квадрате, если результат не выходит за диапазон вещественных чисел.

А эти предописанные* функции дают вещественный результат при целом или вещественном аргументе:

sin(X)	дает синус X; X выражено в радианах;
cos(X)	дает косинус X; X выражено в радианах;
arctan(X)	дает выраженное в радианах значение арктангенса от X;
ln(X)	дает значение натурального (с основанием e) логарифма для X, X > 0;
exp(X)	дает значение экспоненциальной функции (т. е. e в степени X);
sqrt(X)	дает значение корня квадратного X, X >= 0.

*Предупреждение***. Хотя вещественный тип и относится к простым типам, тем не менее его не всегда можно употреблять там, где фигурируют другие простые типы (например, ординальные). В частности, к вещественным аргументам нельзя применять функции pred и succ. Нельзя использовать значения вещественного типа при индексировании массивов, для управления в цикле с параметром, для определения базового типа множеств, для индексирования в операторе варианта.

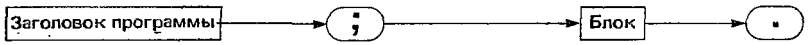
* В некотором смысле такие функции нельзя считать предописанными, т. е. заранее описанными: ведь это вообще не функции. У функций в Паскале тип аргумента строго фиксирован. Раньше они выделялись в совершенно особый класс — стандартные, со своими собственными правилами. — *Примеч. пер.*

** Это предупрежденне осталось от старого описания, где вместо термина «простые» использовался термин «скалярные». Причем скалярные типы даже не классифицировались, а сразу же перечислялись их составляющие. Далее, например, писалось, что в качестве индексов можно использовать значения простого типа. Теперешние же простые типы разбиты на два класса: вещественные и ординальные, и в связи с индексацией упоминаются ординальные. Поэтому предупреждение «повисает в воздухе». — *Примеч. пер.*

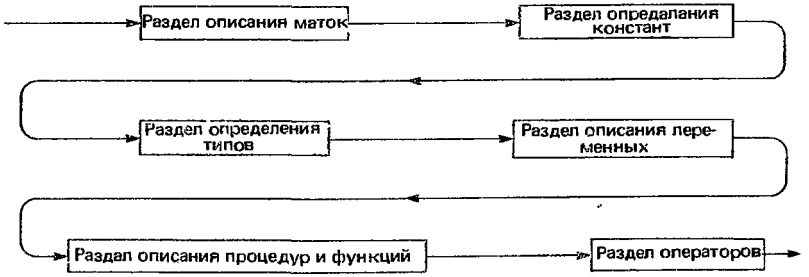
3

ЗАГОЛОВОК ПРОГРАММЫ И РАЗДЕЛ ОПИСАНИЙ

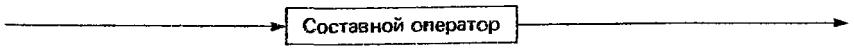
Любая программа состоит из заголовка программы и некоторого блока. Блок содержит раздел описаний, в котором определяются все локальные по отношению к данной программе объекты, и раздел операторов. Он задает действия, которые необходимо выполнять над этими объектами.



Р и с. 3.1. Синтаксическая диаграмма для *Программы*



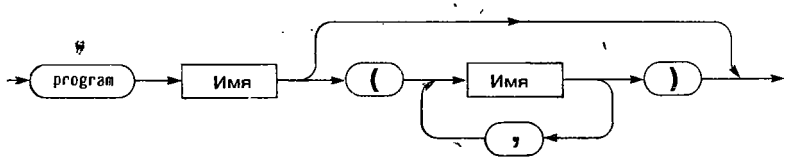
Р и с. 3.2. Синтаксическая диаграмма для *Блока*



Р и с. 3.3. Синтаксическая диаграмма для *Раздела операторов*

3.1. ЗАГОЛОВОК ПРОГРАММЫ

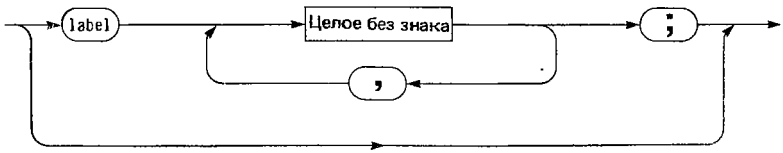
В заголовке программе дается некоторое имя (внутри программы не имеющее какого-либо смысла) и перечисляются ее параметры. Они обозначают объекты, существующие вне программы. Через эти параметры программа взаимодействует с «внешним миром». Такие объекты (обычно — файлы, см. гл. 9) называются *внешними* объектами. Каждый из параметров точно так же, как и обычная локальная переменная, должен быть описан в блоке, составляющем саму программу.



Р и с. 3.4. Синтаксическая диаграмма
для *Заголовка программы*

3.2. РАЗДЕЛ ОПИСАНИЯ МЕТОК

Любой оператор программы можно маркировать, поставив перед ним через двоеточие метку (тем самым появляется возможность ссылаться на эту метку в операторе перехода). Однако такая метка, прежде чем она будет использована, должна быть описана в *разделе описания меток*. Этот раздел начинается со слова label и в общем случае имеет следующий вид:



Р и с. 3.5. Синтаксическая диаграмма
для *Раздела описания меток*

Определяемая метка представляет собой целое число без знака, лежащее в диапазоне от 0 до 9999.

Пример:

```
label 13, 00100, 99;
```

3.3. РАЗДЕЛ ОПРЕДЕЛЕНИЯ КОНСТАНТ

Определение константы вводит имя как синоним некоторой константы. В начале соответствующего раздела стоит слово const, и он в общем случае имеет такой вид:

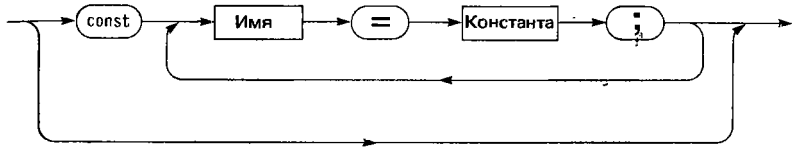


Рис. 3.6. Синтаксическая диаграмма для *Раздела определения констант*

Под константой здесь подразумевается число, имя константы (возможно со знаком), отдельный символ или строка.

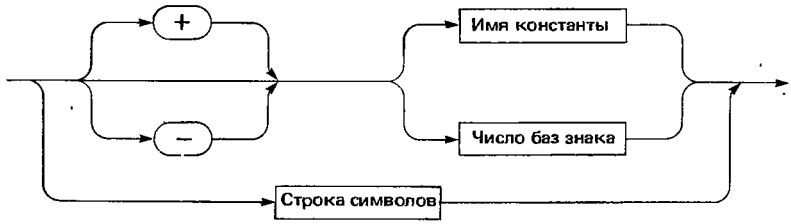


Рис. 3.7. Синтаксическая диаграмма для *Константы*

Использование имен констант делает программу более «читаемой» и способствует улучшению ее документируемости. Кроме того, это позволяет программисту сгруппировать в начале программы величины, зависящие от машины или характерные для данного примера: здесь они более заметны и их легче изменить. Тем самым улучшается переносимость программ и их модульность.

Пример:

```
const
   Avogadro   = 6.023E23;
   PageLength = 60;
   Border     = '# * ';
   MyMove     = True;
```

3.4. РАЗДЕЛ ОПРЕДЕЛЕНИЯ ТИПОВ

Тип данных в Паскале можно описывать непосредственно в описании переменных (см. ниже) либо ссылаться на него через *имя типа*. В Паскале есть несколько мест, где тип можно задавать только с помощью имени. В языке не только предусмотрено несколько имен для стандартных типов, но и существует механизм — *определение типа*, позволяющий вводить для представления типа новое имя типа. Раздел, содержащий определения типов, начинается со слова type и в общем случае имеет такой вид:

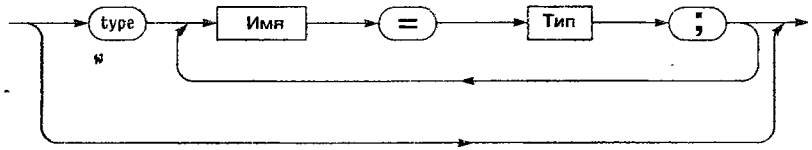


Рис. 3.8. Синтаксическая диаграмма для *Раздела определения типов*

Заметим, что конструкция *Тип* представляет простой, составной или ссылочный тип и выглядит как имя типа, обозначающего уже существующий тип, либо как описание нового типа.

Рис. 3.9. Синтаксическая диаграмма для *Типа*

Примеры определений типов можно найти в последующих разделах.

3.5. РАЗДЕЛ ОПИСАНИЯ ПЕРЕМЕННЫХ

Каждое встречающееся где-либо в программе имя переменной должно быть введено через *описание переменной*. Это описание текстуально должно предшествовать использованию переменной, исключение делается лишь для параметров программы.

Описание переменной вводит имя переменной и связывает с нею тип данных, это делается путем перечисления имен, за которыми

следует тип. Раздел описания переменных начинается со слова var. Общий вид его таков:

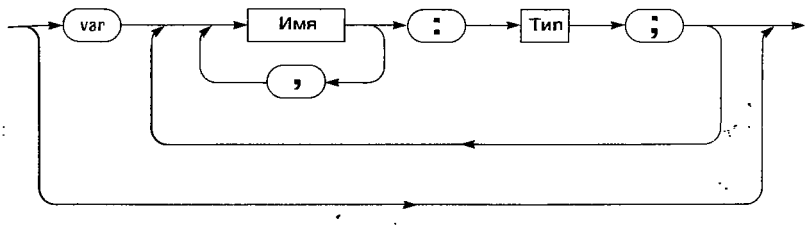


Рис. 3.10. Синтаксическая диаграмма для *Раздела описания переменных*

Пример:

```
var
   Root1, Root2, Root3: Real
   Count, I: Integer;
   Found  Boolean;
   Filler: Char,
```

Каждое имя, перечисленное в заголовке программы в списке параметров и обозначающее внешний объект (обычно файл), кроме имен Input или Output, должно быть описано в разделе описания переменных этой программы. Имена Input или Output, если они перечисляются, автоматически описываются как текстовые файлы (см. гл. 9).

```
program TemperatureConversion(Output);

  { Программа 3.1 — Пример, иллюстрирующий разделы
                    определения констант и типов и описания констант. }

  const
     Bias = 32;  Factor = 1.8;  Low = -20;  High = 39;
     Separator = '   ---';  Blanks = '          ';
  type
     CelciusRange = Low..High {  диапазонный тип — см. гл. 5  };

     var
        Degree: CelciusRange;
```

```
  for Degree := Low to High do
      begin
        Write(Output, Degree, ' C');
        Write(Output, Separator, Round(Degree * Factor + Bias), ' F');
        if odd(Degree) then Writeln(Output)
        else Write(Output, Blanks)
      end;
  Writeln(Output)
end .
```

Дает в качестве результатов:

```
 -20 C  ---   -4 F      -19 C  ---   -2 F
 -18 C  ---    0 F      -17 C  ---    1 F
 -16 C  ---    3 F      -15 C  ---    5 F
 -14 C  ---    7 F      -13 C  ---    9 F
 -12 C  ---   10 F      -11 C  ---   12 F
 -10 C  ---   14 F       -9 C  ---   16 F
  -8 C  ---   18 F       -7 C  ---   19 F
  -6 C  ---   21 F       -5 C  ---   23 F
  -4 C  ---   25 F       -3 C  ---   27 F
  -2 C  ---   28 F       -1 C  ---   30 F
   0 C  ---   32 F        1 C  ---   34 F
   2 C  ---   36 F        3 C  ---   37 F
   4 C  ---   39 F        5 C  ---   41 F
   6 C  ---   43 F        7 C  ---   45 F
   8 C  ---   46 F        9 C  ---   48 F
  10 C  ---   50 F       11 C  ---   52 F
  12 C  ---   54 F       13 C  ---   55 F
  14 C  ---   57 F       15 C  ---   59 F
  16 C  ---   61 F       17 C  ---   63 F
  18 C  ---   64 F       19 C  ---   66 F
  20 C  ---   68 F       21 C  ---   70 F
  22 C  ---   72 F       23 C  ---   73 F
  24 C  ---   75 F       25 C  ---   77 F
  26 C  ---   79 F       27 C  ---   81 F
  28 C  ---   82 F       29 C  ---   84 F
  30 C  ---   86 F       31 C  ---   88 F
  32 C  ---   90 F       33 C  ---   91 F
  34 C  ---   93 F       35 C  ---   95 F
  36 C  ---   97 F       37 C  ---   99 F
  38 C  ---  100 F       39 C  ---  102 F
```

3.6. РАЗДЕЛ ОПИСАНИЯ ПРОЦЕДУР И ФУНКЦИЙ

Прежде чем оно будет использовано, каждое имя процедуры или функции должно быть описано. Описание процедуры и функции имеет вид программы — заголовок, за которым идет блок (детали и примеры приводятся в гл. 11). Процедура представляет собою подпрограмму и активируется (начинает выполняться) через оператор процедуры. Функция — это тоже подпрограмма, но она выдает некоторое значение — результат и поэтому используется как компонента выражения.

3.7 ОБЛАСТЬ ДЕЙСТВИЯ ИМЕН И МЕТОК

Описание или определение имени (константы, типа, переменной, процедуры, функции) либо метки имеет силу везде в блоке, где находится это описание или определение. Исключением являются лишь *вложенные* (внутренние) блоки, в которых переописываются или переопределяются данные имя или метка. Область, где справедливо определение либо описание метки или имени, называется областью действия этой метки или имени.

Если имя или метка описаны либо определены в блоке программы, то их называют *глобальными*. Если же имя или метка описаны либо определены внутри блока, то они называются *локальными* по отношению к этому блоку. Имя или метка не локальны по отношению к этому блоку, если они описаны или определены в блоке, включающем данный. Примеры можно найти в разд. 4.

Имя нельзя описывать на одном уровне или в одной области действия более одного раза. Следовательно, приведенные ниже описания — неверны:

```
var  X: Integer;
     X: Char;
```

4

КОНЦЕПЦИЯ ДЕЙСТВИЯ

Основное в программе для вычислительной машины — выполняемые ею действия. Это означает, что она должна что-то сделать со своими данными, пусть даже это будет принятие решения не делать ничего! Описывают эти действия *операторы*. Операторы бывают простыми (например, оператор присваивания) или сложными. Синтаксическая диаграмма для конструкции *Оператор* приведена на рис. 4.1.

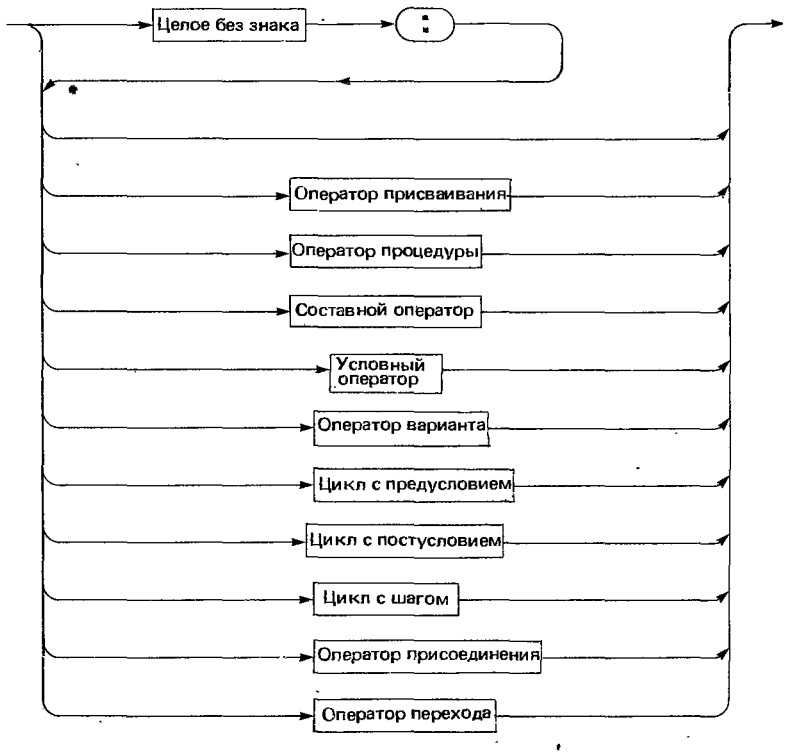
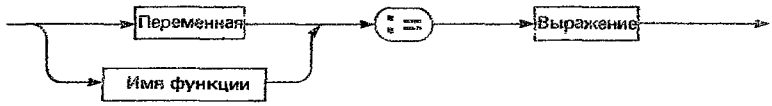


Рис. 4.1. Синтаксическая диаграмма для *Оператора*

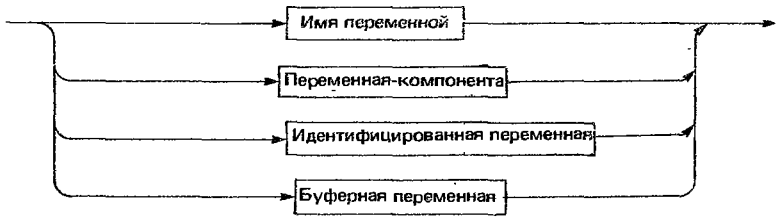
4.1. ОПЕРАТОР ПРИСВАИВАНИЯ И ВЫРАЖЕНИЯ

Самым основным, фундаментальным оператором является *оператор присваивания.* Он указывает, что вновь вычисленное значение необходимо присвоить переменной. Само значение задается выражением. Оператор присваивания имеет вид, определенный диаграммой (рис. 4.2). Символ : = обозначает *присваивание*



Р и с. 4.2. Синтаксическая диаграмма для *Оператора присваивания*

и его не следует путать с операцией отношения =. Оператор «А : = 5» читается так: «текущее значение А заменяется на значение 5», или просто: «А становится равным 5».

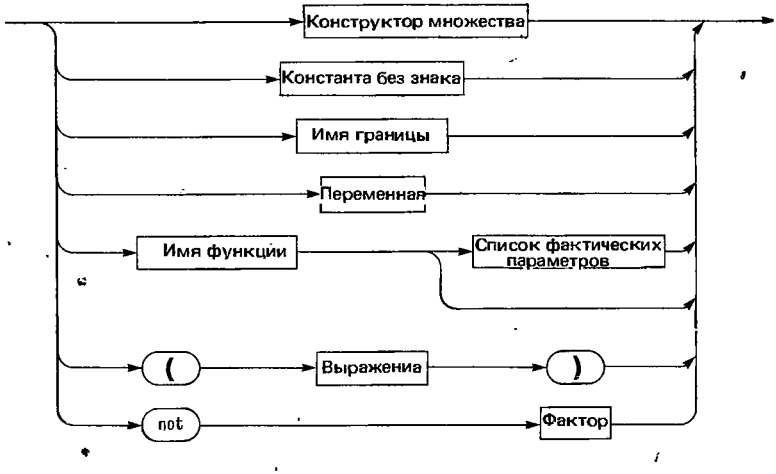


Р и с . 4.3. Синтаксическая диаграмма для *Переменной*

Переменная (рис. 4.3) может быть *полной* переменной, представляющей всю память, выделенную для данного простого, составного или ссылочного типа. Если речь идет о составном типе (см. гл. 6—9), то переменная может быть и *переменной-компонентой* и *буферной переменной,* представляющими одну компоненту памяти, выделенной для данных. В случае ссылочного типа переменная может быть *идентифицированной переменной,* к которой идет косвенное обращение через ссылку.

Выражение состоит из операций и операндов. Операндом может быть константа, переменная, граница параметра-массива (о них речь пойдет в гл. 11) или обозначение функции. (Обозначение функции задает активацию этой функции. В приложении 1 перечи-

сляются предописанные функции, а функции, описываемые программистом, объясняются в гл. 11.)



Р и с. 4.4. Синтаксическая диаграмма для *Фактора*

Выражение задает порядок вычисления значения, основанный на обычном правиле вычисления слева направо и *старшинстве операций.* Выражения состоят из факторов (множителей), термов (слагаемых) и простых выражений.

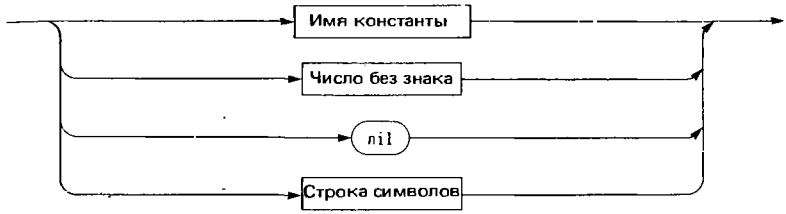
Первыми вычисляются факторы, которые состоят либо из отдельных констант или переменных, либо обозначений функций, либо границ параметров-массивов, либо конструкторов множеств (см. гл. 8). Еще фактор может содержать операцию not, применяемую к другому фактору, представляющему собою логическое значение. Фактор может также включать и выражения, заключенные в скобки; такие выражения вычисляются независимо от предшествующих и последующих операций.

Затем вычисляются термы. Они состоят из последовательности факторов, разделенных мультипликативными операциями (*, /, div, mod, not), или же просто факторов (см. рис. 4.5).

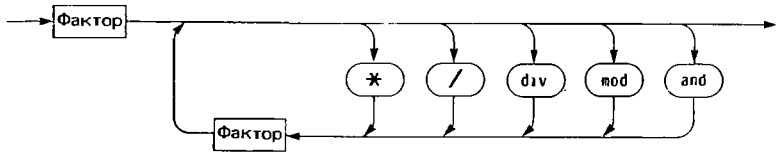
После термов вычисляются простые выражения. Они состоят из последовательности термов, разделенных аддитивными операциями (+, —, or), или же просто термов. Перед первым термом простого выражения может стоять операция изменения знака (+, —) (см. рис. 4.6).

И последним вычисляется выражение. Оно состоит из простого выражения, за которым следует операция отношения (=, <>,

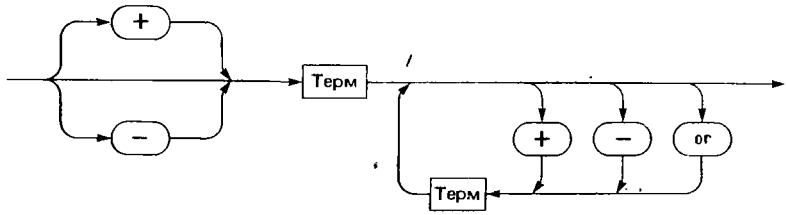
$<=$, $>=$, $>$, in) и другое простое выражение, или же из одного простого выражения (см. рис. 4.7. — 4.8).



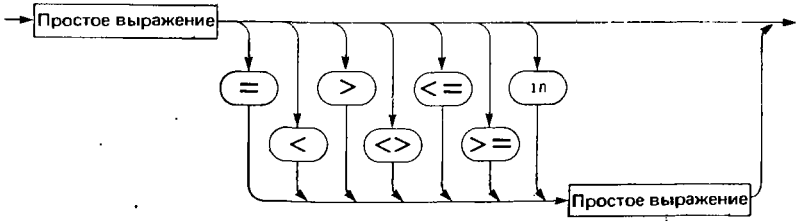
Р и с. 4.5. Синтаксическая диаграмма для *Константы без знака*



Р и с. 4.6. Синтаксическая диаграмма для *Терма*



Р и с. 4.7. Синтаксическая диаграмма для *Простого выражения*



Р и с. 4.8. Синтаксическая диаграмма для *Выражения*

Примеры:

```
2 * 3-4 * 5        = (2*3) - (4*5)  = -14
15 div 4 * 4       = (15 div 4)*4   =  12
80/5/3             = (80/5)/3       =   5.333
4/2 *3             = (4/2)*3        =   6.000
sqrt(sqr(3)+11*5)                   =   8.000
```

Всякий раз, когда возникают затруднения, мы рекомендуем вам обращаться за справками в приведенную ниже таблицу старшинства операций.

Операция	*Классификация*
not	Логическое отрицание (наивысший приоритет)
*, /, div, mod, and	Мультипликативные операции (второй приоритет)
+, —, or	Аддитивные операции (третий приоритет)
=, <>, <, <=, >=, >, in	Операции отношения (низший приоритет)

Полное описание всех операций можно найти в приложении 2.

Для логических выражений характерно то, что их значение может стать известным еще до конца вычисления всего выражения. Предположим, например, что X = 0. Тогда для выражения

(X > 0) and (X < 10)

уже после вычисления первого множителя (фактора) становится ясным результат — false, и второй множитель вычислять нет необходимости. Будет он вычисляться или нет — зависит от реализации. Это означает, что вы всегда должны заботиться об определенности второго фактора, каким бы ни был первый. Следовательно, если предположить, что у массива A индексы лежат в диапазоне от 1 до 10, то приведенный ниже пример ошибочен! (О массивах речь пойдет в гл. 6.)

```
X := 0
repeat X := X + 1 until (X > 10) or (A[X] = 0)
```

(Обратите внимание, что если ни одно значение A[I] не равно нулю, то произойдет обращение к элементу A[11].)

Допустимо присваивание переменным любого типа, за исключением файлового (см. гл. 9). Переменная (или функция) и выражение должны быть *совместимы по присваиванию*. Ниже перечисляются все варианты совместимости по присваиванию.

1. И переменная, и выражение относятся к одному типу, за исключением файлового типа (см. гл. 9) или случая, когда к файловому типу относится какая-либо из компонент составного типа.

2. Переменная относится к типу Real, а выражение — к целому типу.

3. Переменная и выражение относятся к одному или разным диапазонам одного ординального типа (см. гл. 5), и значение выражения лежит внутри замкнутого интервала, определяемого типом переменной.

4. Переменная и выражение относятся к одному множественному типу (см. гл. 8) или множественным типам, базовые типы которых в свою очередь относятся к одному ординальному типу или диапазонам одного ординального типа. Оба типа должны быть либо неупакованными, либо оба — упакованными. Значение выражения должно быть значением, относящимся к типу переменной.

5. Переменная и выражение относятся к строковому типу (см. разд. 6.2) с одинаковым числом элементов.

Примеры присваиваний:

```
Root1 := Pi*X/Y
Root2 := -Root1
Root3 := (Root1 + Root2)*(1.0 + Y)
Danger := Temp > VaporPoint
Count  := Count + 1
Degree := Degree + 10
SqrPr  := sqr(pr)
Y      := sin(X) + cos(Y)
```

4.2. ОПЕРАТОР ПРОЦЕДУРЫ

Еще одним видом простых операторов являются *операторы процедуры;* они активируют поименованную процедуру, представляющую собой некоторую подпрограмму, задающую последовательность действий, которые необходимо выполнить над данными. Например, везде в нашем руководстве мы используем процедуры Read, Readln, Write и Writeln, выполняющие работы по вводу и выводу. Детально операторы процедуры разбираются в гл. 11.

4.3. СОСТАВНОЙ ОПЕРАТОР И ПУСТОЙ ОПЕРАТОР

Составной оператор предусматривает выполнение входящих в него операторов-компонент в порядке их написания. Служебные

слова begin и end выступают в качестве операторных скобок. Обратите внимание, что раздел операторов или «тело» программы есть один составной оператор.

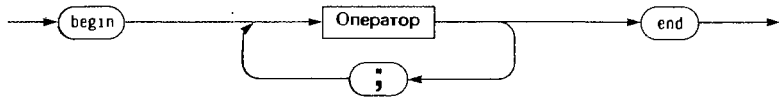


Рис. 4.9. Синтаксическая диаграмма для *Составного оператора*

```
program BeginEndExample(Output);

  { Программа 4.1 — Иллюстрация составного оператора. }

  var
    Sum: Integer;

begin
  Sum := 3 + 5;
  Writeln(Output, Sum, -Sum)
end .
```

Дает в качестве результата:

```
8       -8
```

В Паскале точка с запятой используется как разделитель между операторами, а не заканчивает оператор, т. е. она не входит в оператор. Точные правила, регламентирующие употребление точки с запятой, нашли отражение в синтаксисе, приведенном в приложении 4. Если кто-либо в примере поставит точку с запятой после второго оператора в программе 4.1, то будет считаться, что между этим знаком и словом end находится пустой оператор (не предусматривающий каких-либо действий). Нет ничего страшного в появлении в этом месте пустого оператора. Однако неправильное употребление точки с запятой может приводить и к неприятностям (см. пример с условным оператором из разд. 4.5).

4.4. ОПЕРАТОРЫ ПОВТОРЕНИЯ (ЦИКЛЫ)

Операторы повторения (циклы) предусматривают выполнение некоторых операторов несколько раз. Если число повторений из-

вестно заранее (до начала повторений), то в такой ситуации лучше воспользоваться оператором цикла с параметром. В других же случаях следует использовать операторы цикла с предусловием и с постусловием.

4.4.1. Оператор цикла с предусловием

Такой оператор строится по следующей схеме:

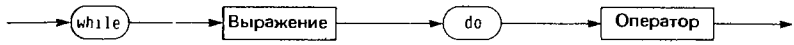


Рис. 4.10. Синтаксическая диаграмма для *Цикла с предусловием*

Оператор, идущий за словом do, выполняется нуль или более раз. Выражение, управляющее повторениями, должно быть логического типа. Это выражение вычисляется до выполнения указанного оператора. Если оно дает значение true, оператор выполняется, в противном случае выполнение всего оператора цикла с предусловием заканчивается. Поскольку выражение вычисляется при каждой итерации, его следует делать насколько возможно простым.

Программа 4.3 возводит вещественное значение X в степень Y, где Y — неотрицательное целое. Более простую и, очевидно, верную версию можно получить, убрав внутренний оператор цикла с предусловием: в этом случае переменная Result получается путем Y-кратного умножения X. Обратите внимание на инвариант цикла: Result*power(Base,Exponent) = power(X, Y).

Внутренний оператор цикла с предусловием оставляет Result и power(Base,Exponent) инвариантными и, очевидно, улучшает «производительность» алгоритма.

4.4.2. Оператор цикла с постусловием

Цикл с постусловием строится по такой схеме:

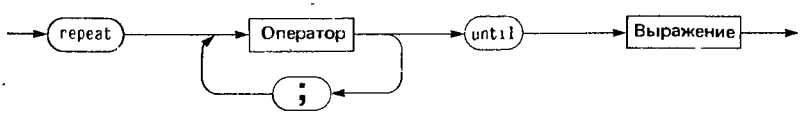


Рис. 4.11. Синтаксическая диаграмма для *Цикла с постусловием*

```
program WhileExample(Input,Output);

   { Программа 4.2 – Вычисление N-й частичной
     суммы гармонического ряда H (N) =1+1/2+1/3+. . .
     + 1/N. Для итерации используется цикл с
     предусловием. }
   var
      N: Integer;
      H: Real;

begin
   Read(Input,N);  Write(Output,N);
   H := 0;
   while N > 0 do
      begin
        H := H + 1/N;  N := N - 1
      end;
   Writeln(Output,H)
end .
```

Дает в качестве результатов:

```
        10 2.928968E+00
```

```
program Exponentiation(Input, Output);

   { Программа 4.3 – Вычисление power (X, Y) – X в степени Y }

   var
      Exponent, Y: Integer;
      Base, Result, X: Real;

begin  Read(Input,X,Y);  Writeln(Output,X,Y);
   Result := 1;  Base := X;  Exponent := Y;
   while Exponent > 0 do
      begin
         { Result*power(Base,Exponent) = power(X,Y),Exponent > 0 }
          while not Odd(Exponent) do
            begin  Exponent := Exponent div 2;  Base := Sqr(Base)
            end;
         Exponent := Exponent - 1;  Result := Result * Base
      end;
```

```
    Writeln(Output,Result) { Result = power(X,Y) }
end
```

Дает в качестве результата:

```
 2.000000E+00          7
 1.280000E+02
```

Последовательность операторов между словами repeat и until выполняется по крайней мере один раз. После каждого выполнения последовательности операторов вычисляется логическое выражение. Повторения продолжаются до тех пор, пока выражение не дает значение true. Так как выражение вычисляется при каждой итерации, его следует делать насколько возможно простым.

```
program RepeatExample(Input,Output);

  {Программа 4.4 — Вычисление N-й частичной суммы гармонического
   ряда H(N) = 1 + 1/2 + 1/3 + . . . 1/N. Для итерации используется
   цикл с постусловием.}

  var
    N: Integer;
    H: Real;

begin
  Read(Input,N);  Write(Output,N);
  H := 0;
  repeat
    H := H + 1/N;  N := N - 1
  until N = 0;
  Writeln(Output,H)
end
```

Дает в качестве результата:

```
         10 2.928968E+00
```

Приведенная выше программа правильно выполняется при $N > 0$. А что произойдет при $N <= 0$? Версия же этой программы с оператором цикла с предусловием верна для любых N, включая $N = 0$.

Обратите внимание, что представляет собой последовательность операторов, выполняемых в операторе цикла с постусловием:

пара скобок begin и end здесь не нужна, однако использование их не будет ошибкой.

4.4.3. Оператор цикла с параметром

Такой оператор предусматривает повторное выполнение некоторого оператора с одновременным изменением по правилу прогрессии значения, присваиваемого *управляющей переменной* (параметру) этого цикла. Оператор цикла с параметром строится по следующей схеме:

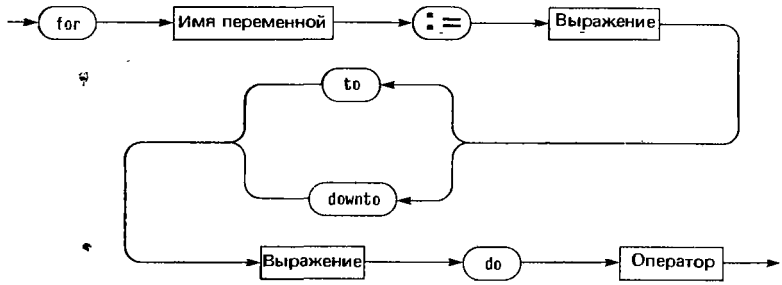


Рис. 4.12. Синтаксическая диаграмма для *Цикла с параметром*

```
program ForExample(Input,Output);

{ Программа 4.5 – Вычисление N-й частичной суммы гармонического
  ряда H(N) = 1 + 1/2 + 1/3 + . . . 1/N. Для итерации используется
  цикл с шагом. }

  var
     I, N: Integer;
     H: Real;
begin
  Read(Input,N);  Write(Output,N);
  H := 0;
  for I := N downto 1 do
     H := H + 1/I;
  Writeln(Output,H)
end .
```

Дает в качестве результатов:

```
         10 2.928968E+00
```

Параметр цикла, идущий следом за словом for, должен относиться к ординальному типу и быть описанным в том же блоке, где появляется сам оператор цикла. Начальное и конечное значения должны относиться к ординальному типу, совместимому с параметром цикла. В операторе-компоненте цикла параметр цикла изменяться не должен, т. е. он не должен фигурировать в качестве переменной в левой части присваивания, в обращениях к процедурам Read или Readln, не должен использоваться как параметр другого оператора цикла с параметром внутри данного оператора цикла с параметром либо внутри процедуры или функции, описанных в данном блоке. Начальное и конечное значения вычисляются лишь единожды. Если при спецификации to (downto) начальное значение больше (меньше) конечного значения, то цикл не выполняется. Если при выполнении оператора-компоненты невозможно присвоить начальное или конечное значение параметру цикла, то такая ситуация считается ошибкой. При нормальном выходе из оператора цикла с параметром управляющая переменная остается неопределенной.

```
program Cosine(Input,Output);
{ Программа 4.6 — Вычисление cos (X) с использованием разложения:
                  cos(X) = 1 - sqr(X)/(2*1)
                           + sqr(X)*sqr(X)/(4*3*2*1) - ... }
  const
    Epsilon = 1e-7;
  var
        Angle: Real   { радианы};
     ASquared: Real   {угол в квадрате};
       Series: Real   {элементы ряда};
         Term: Real   { очередной элемент ряда };
      I, N: Integer   {число вычисляемых косинусов};
     Power: Integer   {порядок очередного элемента};
begin
  Readln(Input,N);
  for I := 1 to N do
    begin
      Readln(Input,Angle);
      Term := 1;  Power := 0;  Series := 1;
      ASquared := Sqr(Angle);
      while Abs(Term) > Epsilon * Abs(Series) do
        begin
          Power := Power + 2;
          Term := -Term * Asquared / (Power * (Power - 1));
          Series := Series + Term
```

```
            end;
          Writeln(Output, Angle, Series, Power div 2
            { =    число элементов   })
        end
    end .
```

Дает в качестве результата:

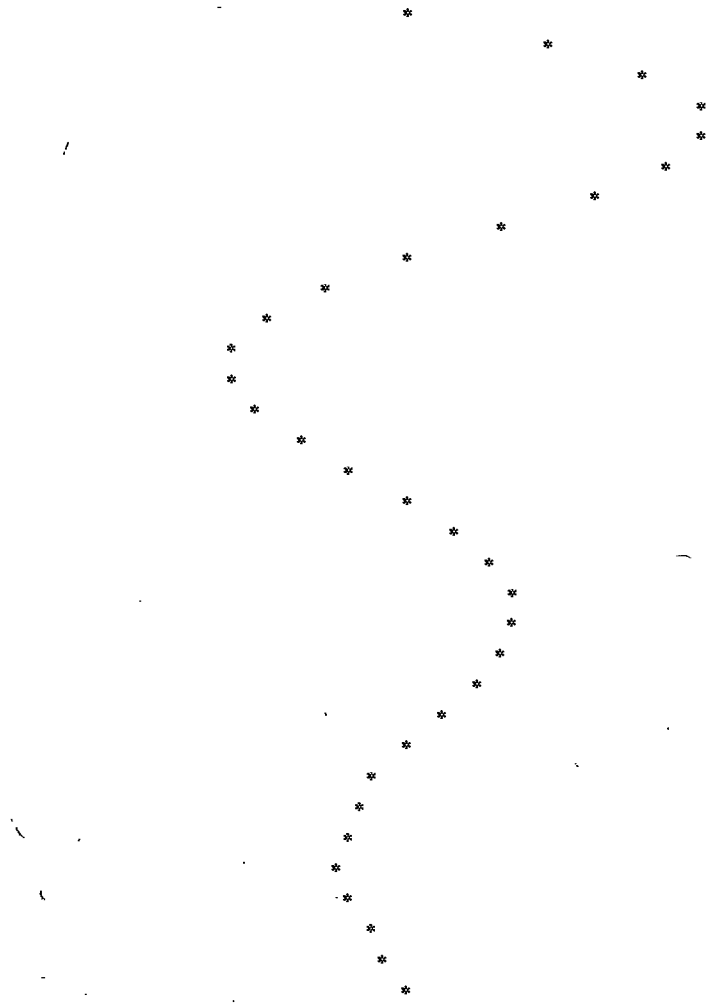
```
 1.534622E-01 9.882478E-01          3
 3.333333E-01 9.449569E-01          4
 5.000000E-01 8.775826E-01          5
 1.000000E+00 5.403023E-01          6
 3.141593E+00-1.000000E+00         10
```

Приведенная ниже программа «рисует» график функции f(X), дающей вещественный результат. Считается, что ось X идет вертикально и на месте соответствующих координат печатается звездочка. Положение звездочки определяется путем вычисления Y = f(X), умножения на некоторый масштабный множитель, округления до следующего целого значения и добавления некоторой константы. Прежде чем напечатать звездочку, в соответствии с полученным результатом «печатается» некоторое количество пробелов.

```
program Graph1(Output);
  {Программа 4.7 — Формирование графического
                    представления функции:
                       f(X) = exp(-X) * sin(2*Pi*X). }
  const
    XLines = 16 {число строк на единицу абсциссы   };
    Scale = 32  { число символов на единицу ординаты };
    ZeroY = 34  { положение оси X };
    XLimit = 32 {размер графика в строках};
  var
    Delta: Real {приращение вдоль абсциссы};
    TwoPi: Real { 2 * Pi = 8 * ArcTan(1.0) };
    X, Y : Real;
    Point: Integer;
    YPosition: Integer;
begin { инициация констант: }
  Delta := 1 / Xlines;
  TwoPi := 8 * ArcTan(1.0);

for Point := 0 to XLimit do
  begin
    X := Delta * Point;  Y := Exp(-X) * Sin(TwoPi * X);
```

```
        YPosition := Round(Scale * Y) + ZeroY;
        repeat
           Write(Output, ' ');  YPosition := YPosition - 1
        until YPosition = 0;
        Writeln(Output, '*')
     end
end.
```

Дает в качестве результата:



В качестве последнего примера рассмотрим такую программу:

```
program SummingTerms(Output);
  { Программа 4.8 – Вычисление ряда
                    1 - 1/2 + 1/3 - ... + 1/9999 - 1/10000
                четырьмя способами:
                1) последовательно слева направо,
                2) слева направо, все положит.,
                   затем – отриц. и вычитание,
                3) последовательно справа налево,
                4) справа налево, все положит.,
                   затем – отриц. и вычитание. }
  var
    SeriesLR,          { сумма, все слева направо }
    SumLRPosTerms,     { сумма положит.,слева направо }
    SumLRNegTerms,     { сумма отриц.,слева направо }
    SeriesRL,          { сумма, все справа налево }
    SumRLPosTerms,     { сумма положит.,справа налево }
    SumRLNegTerms,     { сумма отриц.,справа налево }
    PosTermLR,         { след. положит.,слева направо }
    NegTermLR,         { след. отриц.,слева направо }
    PosTermRL,         { след. положит.,справа налево }
    NegTermRL: Real    { след. положит.,справа налево };
    PairsOfTerms: Integer { счетчик пар членов };

begin
  SeriesLR := 0;  SumLRPosTerms := 0;  SumLRNegTerms := 0;
  SeriesRL := 0;  SumRLPosTerms := 0;  SumRLNegTerms := 0;
  for PairsOfTerms := 1 to 5000 do
    begin
      PosTermLR := 1 / (2 * PairsOfTerms - 1);
      NegTermLR := 1 / (2 * PairsOfTerms);
      PosTermRL := 1 / (10001 - 2 * PairsOfTerms);
      NegTermRL := 1 / (10002 - 2 * PairsOfTerms);
      SeriesLR := SeriesLR + PosTermLR - NegTermLR;
      SumLRPosTerms := SumLRPosTerms + PosTermLR;
      SumLRNegTerms := SumLRNegTerms + NegTermLR;
      SeriesRL := SeriesRL + PosTermRL - NegTermRL;
      SumRLPosTerms := SumRLPosTerms + PosTermRL;
      SumRLNegTerms := SumRLNegTerms + NegTermRL;
    end;
```

```
    Writeln(Output, SeriesLR);
    Writeln(Output, SumLRPosTerms - SumLRNegTerms);
    Writeln(Output, SeriesRL);
    Writeln(Output, SumRLPosTerms - SumRLNegTerms)

end
```

Дает в качестве результатов:

```
 6.930919E-01
 6.931014E-01
 6.930970E-01
 6.930971E-01
```

Почему четыре «идентичные» суммы так отличаются одна от другой?

4.5. ВЫБИРАЮЩИЕ ОПЕРАТОРЫ

Выбирающие операторы предназначены для выделения из составляющих их операторов-компонент одного-единственного оператора, который и выполняется. В Паскале предусмотрены два вида выбирающих операторов: условные операторы и операторы варианта.

4.5.1. Условный оператор

Условный оператор выполняет некоторый оператор только в том случае, если истинно некоторое условие (логическое выражение). Если условие ложно, то либо не выполняется никакой оператор, либо выполняется оператор, идущий следом за словом else.

Условный оператор строится по такой схеме:

Рис. 4.13. Синтаксическая диаграмма для *Условного оператора*

Выражение между словами if и then должно относиться к типу Boolean. Обратите внимание, что первый вид оператора можно

рассматривать как сокращенную запись второго с пустым оператором в качестве альтернативного. Осторожно: перед словом else нет никакой точки с запятой! Поэтому текст:

```
if P then begin S1; S2; S3 end; else S4
```

неверен. Еще более бессмысленным текстом будет такой:

```
if P then; begin S1; S2; S3 end.
```

В этом случае условный оператор управляет пустым оператором, стоящим между словом then и точкой с запятой. Это означает, что составной оператор, идущий за условным оператором, будет выполняться всегда.

Синтаксическая двусмысленность, порождаемая конструкцией:

```
if выражение 1 then if выражение 2 then оператор 1
  else оператор 2
```

разрешается путем ее интерпретации как эквивалентной конструкции:

```
if выражение 1 then
begin if выражение 2 then оператор 1
   else оператор 2
end
```

Читатель должен всегда помнить, что легкомысленное написание условного оператора может обходиться очень дорого. Предположим, у нас есть n *взаимоисключающих условий* $C_1, ..., C_n$, причем каждому соответствует отличное от других действие Si. Пусть P(Ci) — вероятность истинности Ci и пусть $P(Ci) >= P(Cj)$ для $i < j$. В этом случае наиболее эффективная последовательность написания условий такова:

```
if C1 then S1
   else if C2 then S2
      else ..
           else if C(n-1) then S(n-1) else Sn
```

Наступление некоторого условия и выполнение соответствующего действия заканчивает условный оператор, а оставшиеся проверки пропускаются.

Если Found — переменная логического типа, то другим абсурдным и часто употребляемым использованием условного оператора будет:

```
if Key = ValueSought then Found := true else Found := false;
```

Хотя значительно проще было бы написать такой оператор:

```
Found := Key = ValueSought
```

```
program ArabicToRoman(Output);

  { Программа 4.9 — Печать таблицы степеней 2 в "арабском"
                    и "римском" представлении. }

  var
    Rem { остаток },
    Number: Integer;
begin
  Number := 1;
  repeat
    Write(Output, Number, ' ');
    Rem := Number;
    while Rem >= 1000 do
      begin  Write(Output, 'M');  Rem := Rem - 1000  end;
    if Rem >= 900 then
      begin  Write(Output, 'CM');  Rem := Rem - 900  end
    else
      if Rem >= 500 then
        begin  Write(Output, 'D');  Rem := Rem - 500  end
      else
        if Rem >= 400 then
          begin  Write(Output, 'CD');  Rem := Rem - 400  end;
    while Rem >= 100 do
      begin  Write(Output, 'C');  Rem := Rem - 100  end;
    if Rem >= 90 then
      begin  Write(Output, 'XC');  Rem := Rem - 90  end
    else
      if Rem >= 50 then
        begin  Write(Output, 'L');  Rem := Rem - 50  end
      else
        if Rem >= 40 then
          begin  Write(Output, 'XL');  Rem := Rem - 40  end;
    while Rem >= 10 do
      begin  Write(Output, 'X');  Rem := Rem - 10  end;
    if Rem = 9 then
      begin  Write(Output, 'IX');  Rem := Rem - 9  end
    else
      if Rem >= 5 then
        begin  Write(Output, 'V');  Rem := Rem - 5  end
      else
        if Rem = 4 then
          begin  Write(Output, 'IV');  Rem := Rem - 4  end;
    while Rem >= 1 do
```

```
        begin  Write(Output, 'I');  Rem := Rem - 1;  end;
      Writeln(Output);
      Number := Number * 2
    until Number > 5000
end
```

Дает в качестве результата:

```
    1 I
    2 II
    4 IV
    8 VIII
   16 XVI
   32 XXXII
   64 LXIV
  128 CXXVIII
  256 CCLVI
  512 DXII
 1024 MXXIV
 2048 MMXLVIII
 4096 MMMMXCVI
```

Еще раз напоминаем, что каждая «ветвь» условного оператора содержит только по одному оператору. Поэтому, если необходимо выполнять более чем одно действие, следует пользоваться составным оператором.

4.5.2. Оператор варианта

Этот оператор состоит из выражения (селектора) и списка операторов, с каждым из которых сопоставлено одно или несколько постоянных значений, относящихся к типу селектора. Тип селектора должен быть ординальным. Каждое из постоянных значений должно быть сопоставлено самое большее с одним оператором. Оператор варианта выбирает для выполнения оператор, сопоставленный с текущим значением селектора. Если такая константа не была нигде написана, то это ошибка. По окончании выполнения выбранного оператора управление передается в конец всего оператора варианта. Оператор варианта строится по такой схеме:

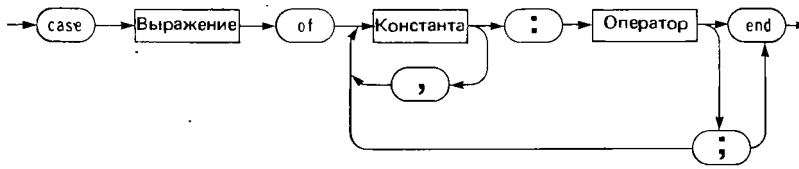


Рис. 4.14. Синтаксическая диаграмма для *Оператора варианта*

Примеры (Предполагается var i: Integer; ch: Char;)

```
case i of
   0: x := 0;
   1: x := sin(x);
   2: x := cos(x),
   3: x := exp(x);
   4: x := ln(x)
end

case ch of
   'a', 'A', 'e', 'E', 'i',
   'I', 'o', 'O', 'u', 'U':
      vowel := vowel + 1;
   '+', '-', '*', '/', '=', '>', '<',
   '.', ',', '"', '?', '!', ':', ';', ''''
      punc := punc + 1
end
```

Замечание. Константы выбора НЕ являются метками (см. разд. 3.2 и 4.7) и на них нельзя ссылаться в операторе перехода. Их упорядоченность произвольная.

Хотя эффективность оператора варианта зависит от реализации, тем не менее в качестве общего правила следует порекомендовать следующее: им нужно пользоваться в случае взаимоисключающих операторов с приблизительно равными вероятностями выбора.

4.6. ОПЕРАТОР ПРИСОЕДИНЕНИЯ

Оператор присоединения употребляется при работе с переменными, относящимися к записному типу (одному из составных типов). О нем речь пойдет в разд. 7.3.

4.7. ОПЕРАТОР ПЕРЕХОДА

Оператор перехода — простой оператор, указывающий, что дальнейшая работа должна продолжаться в другой части текста программы, а именно с того места, где находится метка.

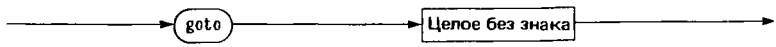


Рис. 4.15. Синтаксическая диаграмма для *Оператора перехода*

Каждая метка:

а) должна появляться в описании метки перед ее использованием в блоке;

б) должна стоять перед *одним и только одним* оператором, входящим в раздел операторов блока;

в) в качестве области действия имеет область, «накрывающую» весь *текст блока*, за исключением каких-либо вложенных блоков, переописывающих данную метку.

Для метки и оператора перехода, указывающего на нее, должно выполняться по крайней мере одно из следующих трех условий:

1. Метка стоит перед оператором, содержащим оператор перехода.

2. Метка стоит перед одним из операторов некоторой последовательности операторов (внутри составного оператора или оператора цикла) и один из операторов этой последовательности содержит оператор перехода.

3. Метка стоит перед оператором из последовательности операторов, представляющих собою раздел операторов блока, содержащего описание процедуры или функции, в которой находится оператор перехода.

Примеры (фрагмент программы):

```
label 1; { блок A }
   ...
   procedure B; { блок B }
      label 3, 5;
   begin
      goto 3;
   3: Writeln('Hello');
   5: if P then begin S; goto 5 end; { while P do S }
      goto 1;
      {это приводит к досрочному окончанию B}
      Writeln('Goodbye')
   end; { block B }

begin
   B;
1: Writeln(' Edsger')
   { "goto 3" в блоке A запрещено }
end { блок A }
```

Переход внутрь сложного оператора извне его запрещается. Следовательно, приведенные примеры неверны.

Примеры неправильных конструкций:

а)
```
for I := 1 to 10 do
    begin S1;
    3: S2
    end;
goto 3
```

б)
```
if B then goto 3;
    ...
if B1 then 3: S
```

в)
```
procedure P:
    procedure Q;
    begin ...
       3: S
    end;
begin ...
    goto 3
end.
```

Операторы перехода следует оставлять для необычных, исключительных ситуаций, когда приходится нарушать естественную структуру алгоритма. Хорошим правилом будет стремление избегать переходов при организации регулярных итераций и условного выполнения операторов, поскольку такие переходы разрушают связь между структурой вычисления и текстуальной (статической) структурой программы. Более того, потеря соответствия между текстуальной структурой и структурой вычисления (статической и динамической) приводит к потере ясности программы и затрудняет задачу верификации. Часто появление в программе на Паскале оператора перехода свидетельствует о том, что программист еще не научился «думать» на Паскале (во все другие языки оператор перехода входит как необходимая компонента)*.

* Заметим, что в Паскале оператор перехода оставлен. Во втором издании книги в этом месте стояло менее категоричное утверждение. — *Примеч. пер.*

5

ПЕРЕЧИСЛЯЕМЫЕ И ДИАПАЗОННЫЕ ТИПЫ

Мы уже познакомились с именами простых предопределенных типов: Boolean, Char, Integer и Real. С помощью таких имен мы можем ссылаться на существующие типы, которые они представляют. Теперь покажем, как с помощью двух механизмов можно порождать совершенно новые ординальные типы: перечисляемые и диапазонные типы. Перечисляемые типы никак не связаны с каким-либо другим типом, это совершенно новые типы; в то время как диапазонные типы — просто подмножества значений некоторого другого, уже существующего типа (ординального).

5.1. ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ

Определение любого перечисляемого типа задает упорядоченное множество значений, перечисляя имена констант, обозначающие эти значения.

Ординальный (порядковый) номер первой из перечисленных констант равен 0, следующей — 1 и т. д.

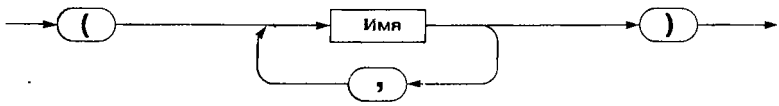


Рис. 5.1. Синтаксическая диаграмма для *Перечисляемого типа*

Пример:

```
type Color = (White, Red, Blue, Yellow, Purple, Green,
              Orange, Black);
   Sex = (Male, Female);
   Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   Operators = (Plus, Minus, Times, Divide);
   Continent = (Africa, Antarctica, Asia, Australia, Europe,
                NorthAmerica, SouthAmerica)
```

Пример неверного определения:

```
type Workday = (Mon, Tues, Wed, Thur, Fri, Sat);
   Free = (Sat, Sun);
```

Неверен он из-за того, что тип константы Sat — двусмыслен.

Вы уже познакомились с предопределенным типом Boolean, который можно было бы определить следующим образом:

type Boolean = (false, true);

Это автоматически предполагает, что для имен констант false и true справедливо соотношение false < true.

Ко всем перечисляемым типам применимы операции отношения (если, конечно, оба операнда одного типа): =, <>, <, <=, >= и >. Порядок устанавливается последовательностью перечисления констант.

Для аргументов, относящихся к ординальным типам, существуют такие предописанные функции:

succ (X)	например	succ(Blue) = Yellow	следующее за X
pred (X)		pred(Blue) = Red	предшествующее X
ord (X)		ord(Blue) = 2	ординальный номер X

Если предположить, что C и C_1 относятся к типу Color, приведенному выше, B — к типу Boolean, а S_1, ..., S_n — некоторые произвольные операторы, то вполне осмысленны следующие операторы:

```
for C := Black downto Red do S1
while (C1 <> C) and B do S1
if C > White then C := pred(C)
case C of
   Red, Blue, Yellow: S1;
   Purple: S2;
   Green, Orange: S3;
   White, Black: S4
end
```

Программа 5.1 иллюстрирует работу с перечисляемыми типами:

```
program DayTime(Output);

  {Программа 5.1 — Пример перечисляемых типов.}
```

```
type
   Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
   When = (Past, Present, Future);

var
   Day: Days;
   Yesterday, Today, Tomorrow: Days;
   Time: When;

begin
  Today := Sun   { В Паскале вводить значение
                   перечисляемого типа нельзя };
  Time := Present;
  repeat
     if Time = Present then  {Вычисление Yesterday}
        begin  Time := Past;
           if Today = Mon then Yesterday := Sun
           else Yesterday := Pred(Today);
           Day := Yesterday
        end
     else
        if Time = Past then {Вычисление Tomorrow}
           begin
              Time := Future;
              if Today = Sun then Tomorrow := Mon
              else Tomorrow := Succ(Today);
              Day := Tomorrow
           end
        else  { Time = Future; Возврат к Present }
           begin  Time := Present;
              Day := Today
           end;
     case Day of
        Mon: Write(Output, 'Monday');
        Tue: Write(Output, 'Tuesday');
        Wed: Write(Output, 'Wednesday');
        Thu: Write(Output, 'Thursday');
        Fri: Write(Output, 'Friday');
        Sat: Write(Output, 'Saturday');
        Sun: Write(Output, 'Sunday')
  end;
```

```
      Writeln(Output, Ord(Time) - 1)
   until Time = Present
end .
```

Дает в качестве результата:

```
Saturday   -1
Monday      1
Sunday      0
```

5.2. ДИАПАЗОННЫЕ ТИПЫ

Этот тип можно определить, указав некоторый *диапазон* значений из любого другого предварительно определенного ординального типа — он называется базовым (host) типом. Диапазон определяется просто указанием наименьшего и наибольшего постоянного значения, входящих в диапазон; причем нижняя граница не должна превышать верхнюю. Выделять диапазон, относящийся к типу Real, нельзя, ведь это не ординальный тип.

Рис. 5.2. Синтаксическая диаграмма для *Диапазонного типа*

Основание диапазонного типа определяет и операции над значениями данного диапазонного типа. Напоминаем: совместимость при присваивании с ординальным типом предполагает, что переменная и выражение относятся к одному типу или к диапазонам одного ординального типа, причем значение выражения находится внутри замкнутого интервала, заданного типом переменной. Пусть, например, дано описание:

```
var A: 1..10; B: 0..30; C: 20..30;
```

Базовый тип для переменных A, B и C — Integer. Следовательно, все присваивания:

```
A := B; C := B; B := C;
```

считаются допустимыми, хотя их выполнение иногда и может приводить к ошибке. Всякий раз, когда в тексте речь пойдет об ординальных типах, будем предполагать, что неявно присутствует дополнение «или диапазон упомянутого типа».

Пример:

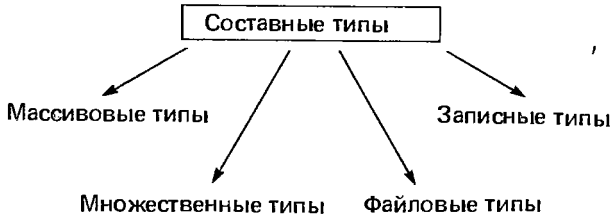
```
type Days = (Mon,Tue,Wed,Thu,Fri,Sat,Sun) {перечисляемый тип};
   Workdays = Mon..Fri { диапазон "дней" };
   Index = 0..63 {диапазон для Integer};
   Letter = 'A'..'Z' {диапазон для Char};
   Natural = 0..MaxInt;
   Positive = 1..MaxInt;
```

Диапазонные типы позволяют формулировать проблему в более наглядной форме. А для реализаторов языка появляется возможность экономить память и проводить во время выполнения программы контроль присваиваний. (В качестве примера употребления диапазонных типов см. программу 6.1.) Если переменная относится, скажем, к типу 0..200, то она может занимать всего один байт, хотя переменная целого типа будет занимать несколько байтов.

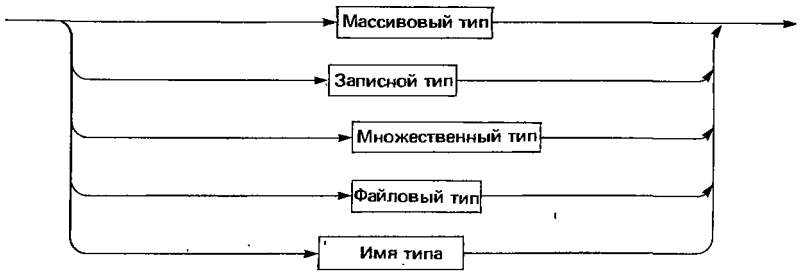
6

ОБЗОР СОСТАВНЫХ ТИПОВ. МАССИВЫ

Простые типы (ординальные и вещественные) — это типы, значения которых не имеют выраженной структуры. Однако в Паскале предусмотрены и другие типы: *составные* (или сложные) и ссылочные. Подобно тому, как сложные операторы состоят из других операторов, составные типы — из других типов. В случае составных типов имеет смысл говорить о *типе* (типах) *компонент* и, что более важно, о методе образования или о структуре сложного типа.



Р и с. 6.1. Схема составных типов данных



Р и с. 6.2. Синтаксическая диаграмма для *Составного типа*

Каковы бы ни были метод образования или структура значений, всегда можно указать предпочтительное внутреннее представление данных. Перед определением типа можно поставить префикс packed (упакованный), указывающий, что транслятор должен эко-

номить память даже за счет дополнительного времени выполнения и увеличения самой программы из-за необходимости операций по упаковке и распаковке. Если пользователь экономит память в ущерб эффективности, то он за это и отвечает. (Фактический эффект от экономии памяти зависит от реализации и в действительности может быть нулевым.)

6.1. МАССИВОВЫЙ ТИП

Любой объект массивового типа (массив) состоит из фиксированного числа компонент (число определяется при введении этого массивового типа). Все компоненты относятся к одному типу, его называют *типом компонент*. Каждая компонента может быть явно обозначена с помощью имени переменной-массива, за которым в квадратных скобках следует *индекс*. Индексы можно вычислять; их тип называется *типом индекса*. Более того, время, требуемое для селекции (доступа) любой компоненты, не зависит от значения селектора (индекса). Поэтому о массивах можно говорить как об объектах, структура которых допускает *случайный* (или *прямой*) *доступ*.

При определении нового массивового типа указывается и тип компонент, и тип индекса. Схема его определения такова:

type A = array [T1] of T2;

где A — имя нового типа, T1 — тип индекса, который должен быть ординальным, а T2 — любой тип.

Массивы позволяют группировать под одним именем несколько переменных, имеющих идентичные характеристики. Описание переменной-массива дает имя всему массиву целиком. Ко всей переменной-массиву применимы две операции: присваивание и выборка (селекция) компоненты. Селекция компоненты осуществляется с помощью задания имени переменной-массива, за которым следует заключенное в квадратные скобки ординальное выражение. К такой переменной-компоненте разрешается применять все операции, допустимые для любых переменных, относящихся к типу компонент данного массивового типа.

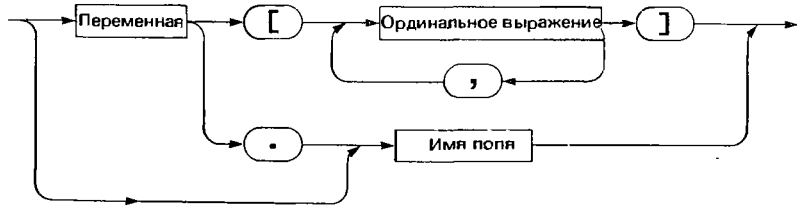


Рис. 6.3. Синтаксическая диаграмма для *Переменной-компоненты*

Примеры описаний переменных:

```
Memory: array [0..Max] of Integer
Sick: packed array [Days] of Boolean
```

Примеры простых присваиваний:

```
Memory[I+J] := X
Sick[Mon] := true
```

(Конечно, мы здесь предполагаем, что вспомогательные переменные описаны.)

Приведенные ниже программы 6.1 и 6.2 иллюстрируют применение массивов. Разберитесь, как можно было бы усовершенствовать программу 6.2, чтобы она строила график более чем одной функции. Сделайте это как с помощью массивов, так и без них.

```
program MinMax(Input,Output);

  { Программа 6.1 – Поиск максимального и минимального
                    числа в заданном списке. }

  const
    MaxSize = 20;

  type
    ListSize = 1..MaxSize;

  var
    Item: ListSize;
    Min, Max, First, Second: Integer;
    A: array [ListSize] of Integer;

begin
  for Item := 1 to MaxSize do
    begin Read(Input, A[Item]); Write(Output, A[Item] :4) end;
  Writeln(Output);

  Min := A[1]; Max := Min; Item := 2;
  while Item < MaxSize do
    begin First := A[Item]; Second := A[Item+1];
      if First > Second then
        begin
          if First > Max then Max := First;
          if Second < Min then Min := Second
        end
```

```
        else
           begin
              if Second > Max then Max := Second;
              if First < Min then Min := First
           end;
        Item := Item + 2
     end;
   if Item = MaxSize then
      if A[MaxSize] > Max then Max := A[MaxSize]
      else
         if A[MaxSize] < Min then Min := A[MaxSize];
   Writeln(Output, Max, Min)
end .
```

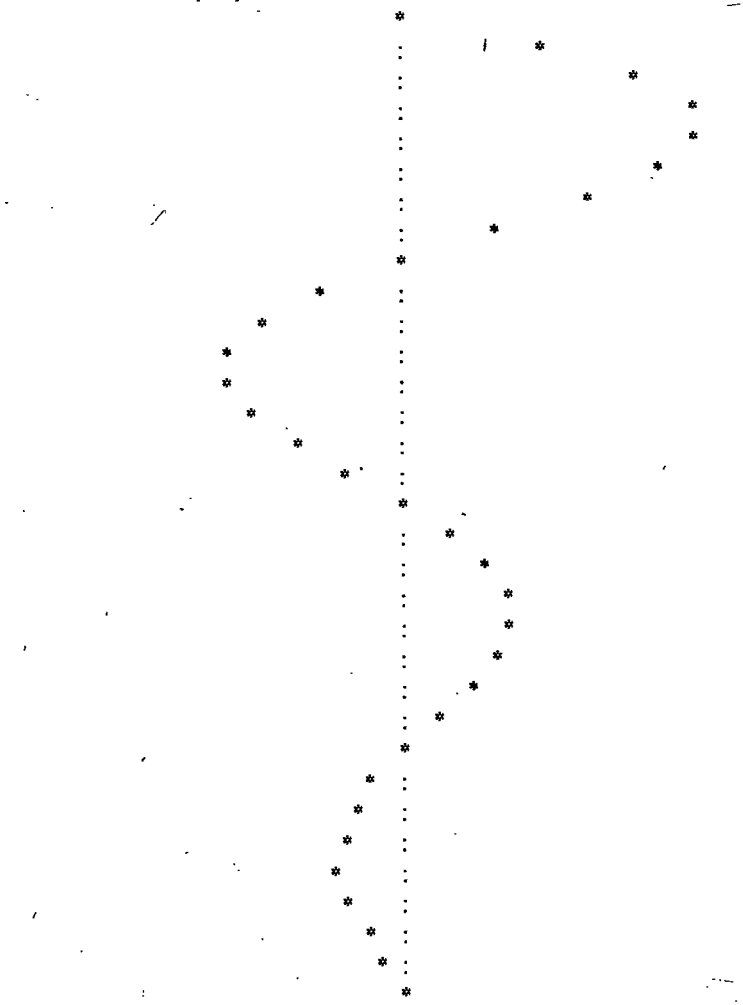
Дает в качестве результата:

```
35  68  94   7  88  -5  -3  12  35   9  -6  -3   0  -2   74  88  52  43   5   4

       94         -6
```

```
program Graph2(Output);
{ Программа 6.2 — формирование графического представления
                  (с осью X) функции:
                  f(X) = exp(-X) * sin(2*Pi*X)    }
const
   XLines = 16 { число строк на единицу абсциссы  };
   Scale = 32  { число символов на единицу ординаты };
   ZeroY = 34  { положение оси X };
   XLimit = 32 {размер графика в строках};
   YLimit = 68 { размер графика в символах};
type
   Domain = 1..YLimit;
var
   Delta: Real { приращение вдоль абсциссы};
   TwoPi: Real { 2 * Pi = 8 * ArcTan(1.0) };
   X, Y:  Real;
   Point: Integer;
   Plot, YPosition, Extent: Domain;
   YPlot: array [Domain] of Char;
begin  { инициация констант: }
   Delta := 1 / Xlines;
   TwoPi := 8 * ArcTan(1.0);

   for Plot := 1 to Ylimit do
      YPlot[Plot] := ' ';
   for Point := 0 to XLimit do
      begin
         X := Delta * Point;  Y := Exp(-X) * Sin(TwoPi * X);
```

```
        YPlot[ZeroY] := ':';
        YPosition := Round(Scale * Y) + ZeroY;
        YPlot[YPosition] := '*';
        if YPosition < ZeroY then Extent := ZeroY
        else Extent := YPosition;
        for Plot := 1 to Extent do Write(Output, YPlot[Plot]);
        Writeln(Output);  YPlot[YPosition] := ' '
      end
end .
```

Дает в качестве результата:



Поскольку T2 может быть любым типом, то компоненты массива могут быть и составного типа. В частности, если T2 опять задает массив, то про исходный массив говорят, что он многомерный. Следовательно, многомерный массив можно описать так:

```
var M: array [A..B] of array [C..D] of T;
```

В этом случае через M[I] [J] обозначается J-я компонента (типа T) I-й компоненты массива M. Обычно для многомерных массивов удобнее пользоваться такой сокращенной формой описания:

```
var M: array [A..B,C..D] of T;
```

а компоненты обозначать через M[I, J].

Массив M можно рассматривать как матрицу и говорить, что M[I, J] относится к компоненте, находящейся в J-м столбце I-й строки этой матрицы.

Дело не ограничивается двумерными массивами: T снова может быть составным типом. В общем случае схема (сокращенного) описания такова:

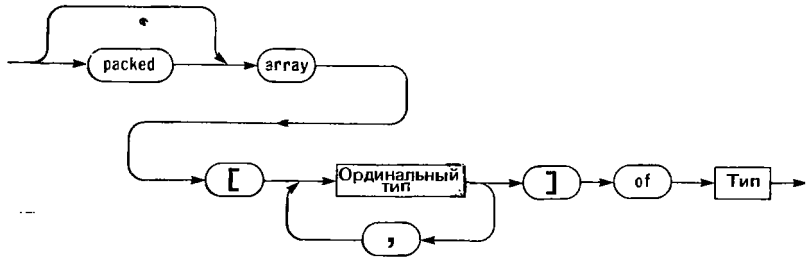


Рис. 6.4. Синтаксическая диаграмма для *Массивового типа*

Если задано n типов индексов, то массив называется *n-мерным,* а его компоненты обозначаются с помощью n индексных выражений. Если есть переменные-массивы A и B одного типа, то присваивание:

```
A := B
```

возможно, если массивы допускают покомпонентное присваивание

```
A[i] := B[i]
```

(для всех i, относящихся к типу индекса), и представляет собою сокращенную запись такого покомпонентного присваивания.

```
program MatrixMul(Input,Output);

  {Программа 6.3 — Умножение матриц.}

  const
    M = 4;  P = 3;  N = 2;

  var
    I: 1..M;
    J: 1..N;
    K: 1..P;
    Sum, Element: Integer;
    A: array [1..M, 1..P] of Integer;
    B: array [1..P, 1..N] of Integer;
    C: array [1..M, 1..N] of Integer;

begin
  {  Начальные значения для A и B.  }
  for I := 1 to M do
    begin
      for K := 1 to P do
        begin  Read(Input,Element);  Write(Output,Element);
          A[I,K] := Element
        end;
      Writeln(Output)
    end;
  Writeln(Output);
  for K := 1 to P do
    begin
      for J := 1 to N do
        begin  Read(Input,Element);  Write(Output,Element);
          B[K,J] := Element
        end;
      Writeln(Output)
    end;
  Writeln(Output);
  {Умножение A на B, C — результат}
  for I := 1 to M do
    begin
      for J := 1 to N do
        begin  Sum := 0;
          for K := 1 to P do
            Sum := Sum + A[I,K] * B[K,J];
          C[I,J] := Sum;  Write(Output,Sum)
        end;
```

```
        Writeln(Output)
    end;
  Writeln(Output)
end
```

Дает в качестве результата:

```
 1        2        3
-2        0        2
 1        0        1
-1        2       -3

-1        3
-2        2
 2        1

 1       10
 6       -4
 1        4
-9       -2
```

Обратите внимание, что в приведенной выше программе для массивов A, B и C типы индексов фиксированы. Если бы мы захотели написать обобщенную подпрограмму перемножения матриц для библиотеки, то нам потребовался бы некоторый механизм согласования типов индексов. В Паскале для таких целей предусмотрены совмещаемые параметры-массивы (см. разд. 11.1.2). Программа 11.4 (MatrixMul2) иллюстрирует использование таких параметров.

6.2. СТРОКОВЫЕ ТИПЫ

Строки уже ранее были определены как последовательности символов, заключенные в апострофы (см. разд. 1.5). Строки, состоящие из одного-единственного символа, представляют собою константы стандартного типа Char (см. разд. 2.4). Строки, состоящие из N символов ($N > 1$), считаются константами типа, определяемого следующим образом:

```
packed array [1..N] of Char
```

Такой тип называется *строковым*.

Если переменная-массив A и выражение E относятся к различным строковым типам, но содержат одно и то же число компонент, допустимо такое присваивание:

```
A := E
```

Аналогично и операции сравнения (=, <>, <, >, <= и >=) можно использовать для сравнения двух строк, если в них одинаковое число компонент, причем упорядоченность «начинается» с первого элемента (A[1]) и определяется порядком, существующим для предописанного типа Char.

6.3. УПАКОВКА И РАСПАКОВКА

Доступ к отдельным компонентам упакованных массивов часто обходится довольно дорого и зависит от ситуации и особенностей конкретной реализации. Поэтому иногда следует упаковывать или распаковывать уже упакованные массивы с помощью одной операции. Это можно сделать с помощью предописанных процедур преобразования Pack и Unpack. Предположим, U — неупакованная переменная-массив, относящаяся к типу:

```
array [A..D] of T {T не может быть типом, содержащим файловый тип},
```

а P — упакованный массив типа

```
packed array [B..C] of T
```

причем ord(D) — ord(A) >= ord(C) — ord(B). Тогда

```
Pack(U, I, P)
```

означает, что речь идет об упаковке части U, начиная с компоненты I, в P. Обращение же

```
Unpack(P, U, I)
```

означает распаковку P в U, начиная с компоненты I.

7

ЗАПИСНЫЕ ТИПЫ

По-видимому наиболее гибким механизмом построения данных являются записные типы. Концептуально любой записной тип задает некоторый образ (шаблон) структуры объектов данного типа, каждая часть которой может иметь совершенно различные характеристики. Предположим, например, что нам необходимо записать информацию о человеке. Мы знаем его имя, номер страхового полиса, пол, дату рождения, число иждивенцев и семейное положение. Если человек женат (замужем) или вдов, то указывается дата свадьбы (последней); если разведен, то указывается дата развода (последнего) и сообщается, первый развод или нет; если — одинокий, то можно ничего не указывать. Вся эта информация может быть выражена одной «записью», к частям которой допускается отдельное обращение.

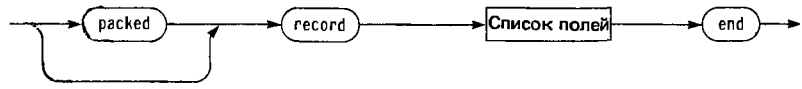
7.1. ФИКСИРОВАННЫЕ ЗАПИСИ

Более формально запись — это некоторое построение, состоящее из фиксированного числа компонент, называемых *полями.* В отличие от массивов компоненты могут относиться к различным типам и непосредственно индексировать их с помощью выражений нельзя. В определении записного типа для каждой из компонент задаются ее тип и имя, *имя поля,* обозначающее эту компоненту. Область действия имени поля — самая внутренняя запись, в которой оно определяется. Целиком ко всей переменной-записи применимы лишь две операции: присваивание и селекция компоненты (выделение).

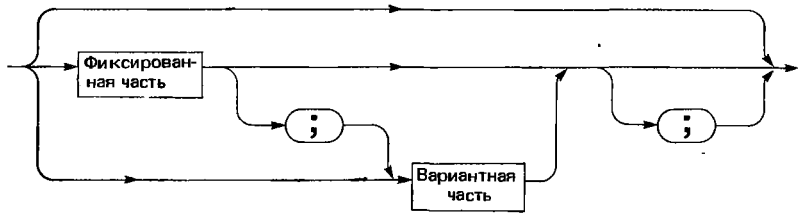
Для того чтобы тип выбранной компоненты был очевиден из текста программы (без ее выполнения), селектор для записей состоит не из вычисляемого значения индекса, а представляет собою постоянное имя поля.

В качестве примера предположим, что мы хотим проводить вычисление с комплексными числами вида a + bi, где a и b — вещественные числа, а i — корень квадратный из — 1. Никакого предопределенного типа complex нет. Однако мы легко можем определить записной тип для представления комплексных чисел. Такие

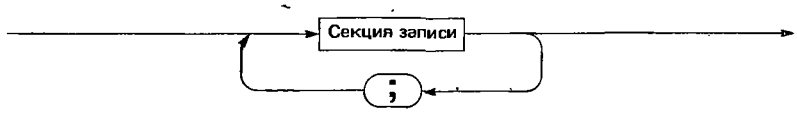
записи должны состоять из двух полей, причем оба поля типа Real: одно — действительная часть, другое — мнимая. При определении типа в данном случае используются следующие синтаксические конструкции:



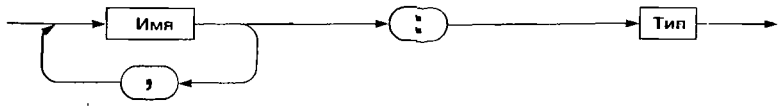
Р и с. 7.1. Синтаксическая диаграмма для *Записного типа*



Р и с. 7.2. Синтаксическая диаграмма для *Списка полей*



Р и с. 7.3. Синтаксическая диаграмма для *Фиксированной части*



Р и с. 7.4. Синтаксическая диаграмма для *Секции записи*

С помощью этих правил мы можем составить такие определение и описание:

```
type Complex = record Re,Im: Real end;
var Z: Complex;
```

здесь Complex — имя типа, Re и Im — имена полей, а Z — переменная типа Complex. Следовательно, Z представляет собою запись, содержащую две компоненты или поля (см. программу 7.1).

Для обращения к компоненте записи следом за именем записи ставится точка, а затем — имя соответствующего поля (см. рис. 6.3). Например, значение 5 + 3i переменной Z присваивается так*:

```
Z.Re := 5:
Z.Im := 3
```

Аналогично определению типа Complex можно определить и тип, соответствующий понятию «дата».

```
Date = packed record
                Year: 1900..2100;
                Mo: (Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec);
                Day: 1..31
            end
```

Заметим, что среди значений этого типа присутствует и дата «31 апреля». Можно определить тип для понятия «игрушка» (toy):

```
Toy = record
        Kind: (Ball, Top, Boat, Doll, Blocks,
               Game, Model, Book);
        Cost: Real;
        Received: Date;
        Enjoyed: (Alot, Some, Alittle, None);
        Broken, Lost: Boolean
      end
```

Можно определить и «домашнее задание»:

```
Assignment = packed record
                      Subject:(History, Language, Lit',
                               Math, Psych, Science);
                      Assigned: Date;
                      Grade: 0..4;
                      Weight: 1..10
                    end
```

* Формально такая запись может быть и верна (без последней «точки с запятой»), но производит впечатление неряшливой. — *Примеч. пер.*

```
program ComplexArithmetic(Output);

   { Программа 7.1 — Пример операций над комплексными числами. }

   const
      Increment = 4;

   type
      Complex =
         record
            Re, Im: Real
         end;

   var
      X, Y: Complex;
      Pair: Integer;

begin
   X.Re := 2;  X.Im := 5;  { инициация X }
   Y := X;                 { инициация Y }

   for Pair := 1 to 5 do
      begin
         Writeln(Output, 'X = ', X.Re :5:1, X.Im :5:1, 'i');
         Writeln(Output, 'Y = ', Y.Re :5:1, Y.Im :5:1, 'i');

         {X + Y}
         Writeln(Output, 'Sum = ', X.Re + Y.Re :5:1,
                         X.Im + Y.Im :5:1, 'i');

         {X * Y}
         Writeln(Output, 'Product = ', X.Re * Y.Re - X.Im*Y.Im :5:1,
                           X.Re * Y.Im + X.Im * Y.Re :5:1, 'i');
         Writeln(Output);

         X.Re := X.Re + Increment;
         X.Im := X.Im - Increment
      end
end .
```

Дает в качестве результатов:

```
X =   2.0  5.0i
Y =   2.0  5.0i
Sum =   4.0 10.0i
Product = -21.0 20.0i
```

```
X =   6.0  1.0i
Y =   2.0  5.0i
Sum =   8.0  6.0i
Product =   7.0 32.0i

X =  10.0 -3.0i
Y =   2.0  5.0i
Sum =  12.0  2.0i
Product =  35.0 44.0i

X =  14.0 -7.0i
Y =   2.0  5.0i
Sum =  16.0 -2.0i
Product =  63.0 56.0i

X =  18.0-11.0i
Y =   2.0  5.0i
Sum =  20.0 -6.0i
Product =  91.0 68.0i
```

Если некоторая запись сама входит в состав данных более сложной структуры, то эта структура отражается и на имени записи. Предположим, что нам надо записать дату самой последней прививки оспы для каждого из членов семьи. Членов семьи можно представить с помощью некоторого перечисляемого типа, а соответствующие даты хранить в массиве записей:

```
type FamilyMember = (Father, Mother, Child1, Child1, Child3);
var VaccinationDate: array [FamilyMember] of Date;
```

Формирование даты после этого можно записать так:

```
VaccinationDate[Child3].Mo := Apr;
VaccinationDate[Child3].Day := 23;
VaccinationDate[Child3].Year := 1973
```

7.2. ВАРИАНТНЫЕ ЗАПИСИ

Иногда в запись необходимо включать информацию, зависящую от другой, уже включенной в эту же запись информации. Поэтому мы можем определить вариантный записной тип, где есть дополнительные поля, зависящие от значений других полей.

Синтаксис записного типа предусматривает *вариантную часть*, рассчитанную на то, что можно задавать тип, содержащий определение нескольких вариантов структуры. Это означает, что разные переменные, хотя они и относятся к одному типу, могут иметь отличающуюся друг от друга структуру. Различие может касаться как числа компонент, так и их типов.

Каждый вариант характеризуется задаваемым в скобках списком описаний присущих ему компонент. Перед списком находятся одна или несколько констант, а перед всей группой списков стоит заголовок варианта, где указывается тип этих констант (т. е. тип, на основании значений которого мы различаем варианты).

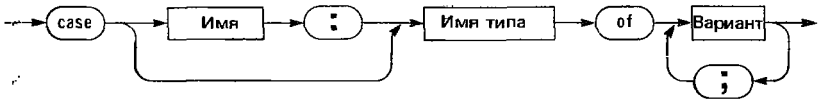


Рис. 7.5. Синтаксическая диаграмма для *Вариантной части*

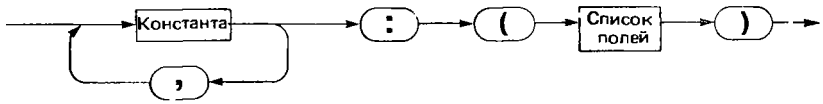


Рис. 7.6. Синтаксическая диаграмма для *Варианта*

Предположим, что есть, например, такое описание:

type MaritalStatus = (Married, Widowed, Divorced, Single)

В этом случае человека можно описать с помощью данных следующего типа:

```
type Person =
     record
        { здесь поля общие для всех Person };
        case MaritalStatus of
           Married: ( { поля только для семейных } );
           Single: ( { поля только для одиноких } );
              ...
     end
```

Обратите внимание, что каждому значению, относящемуся к типу, на основании которого идет различение вариантов *(типу признака)*, должен соответствовать один из вариантов. (Это означает, что перед вариантами должны появляться все значения типа

признака.) Скажем, в нашем примере, кроме констант Married и Single, должны появиться и константы Widowed и Divorced.

Обычно некоторая компонента (поле) записи сама указывает, о каком варианте идет речь. Например, в приведенном выше описании типа следовало бы иметь общее для всех вариантов поле.

MS: MaritalStatus

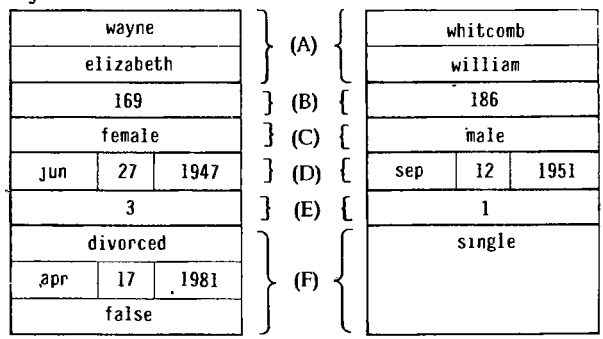
Это частная ситуация, и для нее существует сокращение: описание определяющей вариант компоненты, называемой полем признака, включается в сам заголовок варианта. Например:

case MS: MaritalStatus of

Прежде чем начать определять структуру записи с вариантами, соответствующую, например, типу Person, полезно бывает «выписать» всю необходимую информацию.

1. Человек
   A. Имя (name) — первое, последнее (first, last)
   B. Рост (height) — целое число
   C. Пол (sex) — муж., жен (male, female)
   D. Дата рождения (data of birth) — год, месяц, день (year, month, day)
   E. Число иждивенцев (dependents) — целое число
   F. Семейное положение (marital status):
      если в браке (married) или вдов (widowed):
         а) дата свадьбы (date of marriage) — год, месяц, день (year, month, day)
      если разведен (divorced):
         а) дата развода (date of divorce) — год, месяц, день (year, month, day)
         б) первый развод (first divorce) — нет, да (false, true)
      если одинокий (single)

На рис. 7.7 приведены «образы» двух «простых» людей с различными атрибутами.



Р и с. 7.7. Пример описания двух людей

Теперь определение записного типа Person можно сформулировать так:

```
type String15 = packed array [1..15] of Char;
   Status = (Married, Widowed, Divorced, Single);
   Date = packed record
                  Year: 1900..2100;
                  Mo: (Jan, Feb, Mar, Apr, May, Jun,
                       Jul, Aug, Sep, Oct, Nov, Dec);
                  Day: 1..31;
                end;
   Natural = 0..MaxInt;
   Person = record
              Name: record First, Last: String15 end;
              Height: Natural { centimeters };
              Sex: (Male, Female);
              Birth: Date;
              Depdts: Natural
              case MS: Status of
                 Married, Widowed: (MDate: Date);
                 Divorced: (DDate: Date;
                            FirstD: Boolean);
                 Single: ()
            end { Person };
```

Замечания.

1. Все имена полей должны быть различными, даже если они встречаются в разных вариантах.

2. Если вариант пустой (т. е. поля нет), то он записывается так: C:().

3. Любой список полей может иметь только одну вариантную часть, которая должна следовать за фиксированной частью записи.

4. Каждый вариант может содержать в себе вариантную часть, следовательно, допускаются вложенные варианты.

5. Область действия имен констант перечисляемого типа, вводимых в записном типе, расширяется на вложенные блоки.

При обращении к компонентам записи их имена, по существу, представляют простую линейную последовательность всех имен. Если, например, P — переменная типа Person, то второй, приведенный на рис. 7.7 «образ» порождается такими присваиваниями:

```
P.Name.Last := 'Whitcomb       ';
P.Name.First := 'William       ';
P.Height := 186;
P.Sex := Male;
P.Birth.Year := 1951;
P.Birth.Mo := Sep;
P.Birth.Day := 12;
P.Depdts := 1;
P.MS := Single;
```

7.3. ОПЕРАТОР ПРИСОЕДИНЕНИЯ

Приведенные выше действия, возможно, немного «утомительны», и вы можете сократить их, обратившись к *оператору присоединения*. Фактически такой оператор открывает область действия, содержащую имена полей указанной переменной-записи, так что теперь эти имена могут фигурировать как имена обычных переменных. (Тем самым и транслятору дается возможность оптимизировать уточненный (qualified) оператор). Оператор строится по такой схеме:

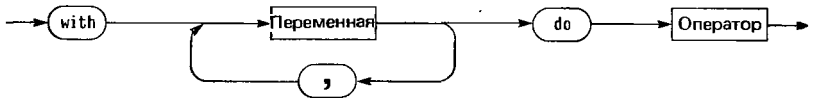


Рис. 7.8. Синтаксическая диаграмма для *Оператора присоединения*

Внутри оператора, входящего в оператор присоединения, поля переменной-записи обозначаются с помощью только имен полей (имя переменной-записи перед ними не указывается).

Приведенный ниже оператор присоединения эквивалентен всей серии предыдущих присваиваний.

```
with P do
  begin
    with Name do
      begin Last := 'Whitcomb
        First := 'William
      end;
    Height := 186;
```

```
    Sex := Male;
    with Birth do
      begin Year := 1951; Mo := Sep; Day := 12 end;
    Depdts := 1;
    MS := Single;
  end
```

Аналогично фрагмент:

```
  var CurrentDate: Date;
  ...
  with Currentdate do
    if Mo = Dec then
      begin Mo := Jan; Year := Year + 1 end
    else Mo := succ(Mo)
```

эквивалентен:

```
  var CurrentDate: Date;
  ...
  if CurrentDate.Mo = Dec then
    begin CurrentDate.Mo := Jan;
      CurrentDate.Year := CurrentDate.Year + 1 end
  else CurrentDate.Mo := succ(CurrentDate.Mo)
```

Пример с вакцинацией теперь может быть записан так:

```
  with VaccinationDate[Child3] do
    begin Year := 1973; Mo := Apr; Day := 23 end
```

При выполнении оператора присоединения ссылки на переменную-запись устанавливаются еще до выполнения внутреннего оператора. Поэтому выполняемые внутри его какие-либо присваивания элементам списка переменных-записей не будут изменять выделенную заранее запись.
Например:

```
  var Who: FamilyMember;
  ...
  Who := Father;
  with VaccinationDate[Who] do
    begin
      Who := Mother;
      Mo := Jul; Day := 7; Year := 1947
    end
```

Оператор присоединения установит поле VaccinationDate [Father].

Вложенные операторы присоединения допускают сокращенную запись.

Оператор такого вида with R1, R2, ..., Rn do S эквивалентен:

```
with R1 do
  with R2 do
    ...
      with Rn do S
```

Таким образом, предыдущий пример с определением человека P можно переписать следующим образом:

```
with P, Name, Birth do
  begin Last := 'Whitcomb       ';
    First := 'William        ';
    Height := 186;
    Sex := Male;
    Year := 1951;
    Mo := Sep;
    Day := 12;
    Depdts := 1;
    MS := Single;
  end { with }
```

А теперь рассмотрим пример, иллюстрирующий правило областей действия для имен полей. Хотя описания:

```
var A: array [2..8] of Integer;
  A: 2..8;
```

недопустимы, так как определение A двусмысленно, тем не менее описание:

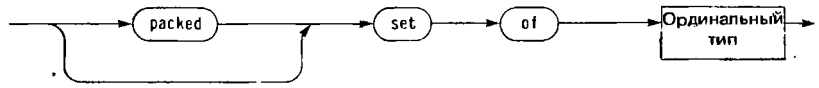
```
var A: Integer;
  B: record A: Real; B: Boolean
    end;
```

вполне корректно, поскольку обозначение для целого A легко отличить от обозначения для вещественного B.A. Аналогично и переменная-запись B легко отличается от логического B.B. Внутри уточненного оператора S из with B do S имена A и B теперь соответственно обозначают компоненты B.A и B.B, а целая переменная с именем A недоступна.

8

МНОЖЕСТВЕННЫЕ ТИПЫ

Множественные типы обеспечивают компактную структуру, в которой сохраняется информация о группах значений, относящихся к ординальному типу; можно узнать, есть ли элемент в группе и какова комбинация этих элементов. Более точно множественный тип определяет множество значений, которое представляет собою множество-степень базового типа, т. е. множество всех подмножеств базового типа, включая и пустое множество. Поэтому одиночное значение множественного типа есть множество, причем элементы этого множества — элементы базового типа. Множество — структура со случайным доступом, все ее элементы относятся к одному базовому типу, который должен быть ординальным типом*.



Р и с. 8.1. Синтаксическая диаграмма для *Множественного типа*

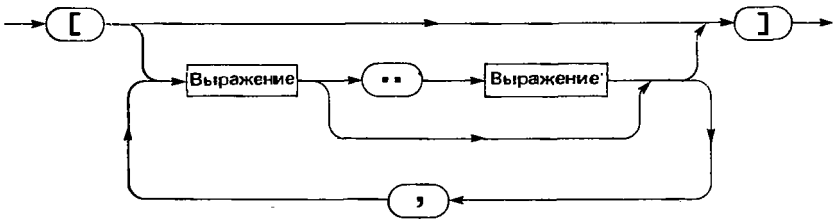
Для полных значений-множеств применяются такие операции: присваивания, традиционные операции над множествами (например, объединение), проверка на равенство и выборка компоненты, позволяющая проверить присутствие элемента в множестве (см. ниже). Реализация Паскаля обычно определяет предельный размер множеств, причем он может быть достаточно небольшим (например, число разрядов в «слове» машины). Этот предел непосредственно связан с диапазоном значений базового типа данного множественного типа.

* Последнее высказывание не имеет смысла, так как доступа к отдельным элементам множества нет. Они даже значения не имеют. — *Примеч. пер.*

8.1. КОНСТРУКТОРЫ МНОЖЕСТВ

Множественное значение можно задать с помощью конструктора множества, в котором содержатся описания элементов множества, отделенные друг от друга запятыми и заключенные в квадратные скобки. Описанием элемента может быть выражение, значение которого и есть элемент, или диапазон вида low..high, где значения выражений low и high представляют собою нижнюю и верхнюю границы группы элементов. Если нижняя граница больше верхней границы группы (т. е. low > high), то никакой элемент не описывается.

Все выражения должны относиться к одному ординальному типу, представляющему собой базовый тип для множественного типа данного конструктора. Конструктор множества [] обозначает пустое множество для любого множественного типа. В конструкторе множества нет полной информации о типе (см. [10]), например, такой как, упаковано множество или нет. Поэтому тип конструктора множества одновременно относится и к упакованному, и неупакованному и выбирается так, чтобы быть совместимым с другими множествами в данном множественном выражении.



Р и с. 8.2. Синтаксическая диаграмма
для *Конструктора множества*

Примеры конструкторов множеств:

```
[13]
[i+j,i-j]
['0'..'9']
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
 'p','q','r','s','t','u','v','w','x','y','z']
```

8.2. ОПЕРАЦИИ НАД МНОЖЕСТВАМИ

Если X — переменная-множество, а E — множественное выражение, то присваивание:

X := E

допустимо только в том случае, если все элементы E относятся к базовому типу X, и, кроме того, типы X и E либо оба упакованы, либо не упакованы. К любым объектам со структурой множества применимы такие операции. Если предположить, что A и B — выражения одного типа, то:

A + B	множество из элементов A и B (объединение)
A * B	множество общих для A и B элементов (пересечение)
A — B	множество элементов A, не входящих в B (разность)

К множественным операндам применимы пять операций отношения. Предположим, A и B — множественные выражения одного типа, а e — ординальное выражение базового типа.

e in A	вхождение в множество; результат true, если e элемент A, иначе — false
A = B	равенство множеств
A <> B	неравенство множеств
A <= B	включение; результат true, если A — собственное или несобственное подмножество B
A >= B	включение; результат true, если B — собственное или несобственное подмножество A

Примеры описаний:

```
type Primary = (Red, Yellow, Blue);
       Color = set of Primary;
var Hue1, Hue2: Color;
   Vowels, Consonants, Letters: set of Char;
   Opcode: set of 0..7;
       Add: Boolean;
        Ch: Char;
```

Примеры присваиваний:

```
Hue1 := [Red]; Hue2 := [];
Hue2 := Hue2 + [succ(Red)]
```

```
Letters := ['A','B','C','D','E','F','G','H','I',
           'J','K','L','M','N','O','P','Q','R',
           'S','T','U','V','W','X','Y','Z'];
Vowels := ['A','E','I','O',U];
Consonants := Letters - Vowels

Add := [2,3] <= Opcode
```

Действия с множествами относительно быстрые и их можно использовать для исключения более сложных проверок. Например, вместо проверки

if (Ch = 'A') or (Ch = 'E') or (Ch = 'I') or (Ch = 'O') or (Ch = 'U') then S

можно написать более простую

if Ch in ['A', 'E', 'I', 'O', 'U'] then S

```
program Convert(Input,Output);

  { Программа 8.1 — Чтение последовательности цифр и
                    преобразование их в целое. Знака нет. }
  var
    Ch: Char;
    Digits: set of '0'..'9';
    Number: Integer;
begin
  Digits := ['0'..'9'] {инициация значения множества};
  Read(Input, Ch);
  Number := 0;
  while Ch in Digits do
    begin
      Number := Number * 10 + Ord(Ch) - Ord('0');
      Writeln(Output, Number);
      Read(Input, Ch)
    end
  { Ch содержит символ, идущий за целым }
end
```

Дает в качестве результата:

```
4
43
432
4321
```

```
program SetOperations(Output);

   { Программа 8.2 – Пример операций над множествами. }

   type
      Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
      Week = set of Days;

   var
      FullWeek, Work, Free: Week;
      Day: Days;

   procedure Check(W: Week); { процедуры вводятся в гл. 11 }

      var D: Days;

   begin
      for D := Mon to Sun do
         if D in W then Write(Output, 'x') else Write(Output, 'o');
      Writeln(Output)
   end { Check };

begin
   Work:= [];  Free := [];  FullWeek := [Mon..Sun];
   Day := Sat;
   Free := [Day] + Free + [Sun];
   Check(Free);
   Work := FullWeek - Free;
   Check(Work);
   if Free <= FullWeek then Write(Output, 'O');
   if FullWeek >= Work then Write(Output, 'K');
   if not (Work >= Free) then Write(Output, ' Jack');
   if [Sat] <= Work then Write(Output, 'Forget it!');
   Writeln(Output)
end
```

Дает в качестве результата:

```
ooooOxx
xxxxxoo
OK Jack
```

8.3. РАЗРАБОТКА ПРОГРАММ

Программирование в смысле построения и формулирования алгоритмов, в общем, сложный процесс, требующий владения определенными приемами и умения учитывать многие детали. Только в редких случаях здесь существует одно-единственное «хорошее» решение. Обычно перед нами так много решений, что выбор оптимального требует основательного анализа не только возможных алгоритмов и машин, но даже и условий, в которых наиболее часто будет использоваться наша программа.

Поэтому конструирование алгоритма должно состоять из последовательности обдумывания, исследования и принятия конструктивных решений. На ранних этапах лучше всего концентрировать внимание на глобальных проблемах и при первом наброске решения не останавливаться на деталях. По мере продвижения процесса задачу можно разбивать на подзадачи, уделять больше внимания деталям определения задачи и особенностям используемых приемов программирования. Такой подход к программированию часто называют структурным программированием [4], или программированием методом последовательных уточнений [2].

Ниже мы покажем процесс создания алгоритма на примере, взятом у Хоара из [4]. Мы его перескажем, естественно, используя понятия, присущие Паскалю.

Нужно найти простые числа, лежащие в диапазоне 2..n, где n >= 2. После сравнения различных алгоритмов мы остановимся на «решете Эратосфена», ибо в этом алгоритме не используются ни умножение, ни деление.

Сначала опишем алгоритм «словами»:

1. Поместим все числа между 2 и n в «решето».
2. Выберем и вынем из «решета» наименьшее из оставшихся в нем чисел.
3. Поместим это число среди «простых» чисел.
4. Переберем и вынем из «решета» все числа, кратные данному.
5. Если «решето» не пустое, то повторим шаги 2—5.

Хотя при выполнении программы в первую очередь проводится инициация переменных, тем не менее при разработке соответствующие операторы пишутся в последнюю очередь. Для того чтобы записать эти действия правильно, требуется исчерпывающее понимание алгоритма. Если нужно сохранять работоспособность программы, то при каждой ее модификации операторы инициации придется корректировать. К сожалению, даже коррекции не всегда достаточно.

Для представления как «решета», так и множества простых чисел Хоар использует множества с элементами от 2 до n. Ниже

дается слегка видоизмененный «набросок» его программы:

```
program Primel;

{Программа 8.3 — Использование для решета Эратосфена множеств }

  const
    N = 10000;

  type
    Positive = 1..MaxInt;

  var
    Sieve, Primes: set of 2..N;
    NextPrime, Multiple: Positive;

begin { инициация }
  Sieve := [2..N];  Primes := [];  NextPrime := 2;
  repeat {поиск очередного простого }
    while not (NextPrime in Sieve) do NextPrime := Succ(NextPrime);
    Primes := Primes + [NextPrime];
    Multiple := NextPrime;
    while Multiple <= N do {исключение}
      begin  Sieve := Sieve - [Multiple];
       . Multiple := Multiple + NextPrime;
      end
  until Sieve = []
end .
```

В качестве упражнения Хоар предлагает переписать программу так, чтобы в множествах были только нечетные числа. Ниже приводится решение; обратите внимание на его связь с первым решением.

```
program Prime2;

  {Программа 8.4 — Использование для решета Эратосфена множеств;
                   только нечетные числа.}

  const
    N = 5000  { N' = N div 2 };

  type
    Positive = 1..MaxInt;
```

```
    var
       Sieve, Primes: set of 2..N;
       NextPrime, Multiple, NewPrime: Positive;

begin  { инициация }
   Sieve := [2..N];  Primes := [];  NextPrime := 2;
   repeat  { поиск очередного простого }
      while not (NextPrime in Sieve) do NextPrime := Succ(NextPrime);
      Primes := Primes + [NextPrime];
      NewPrime := 2 * NextPrime - 1;
      Multiple := NextPrime;
      while Multiple <= N do  { исключение }
         begin  Sieve := Sieve - [Multiple];
            Multiple := Multiple + NextPrime;
         end
   until Sieve = []
end .
```

Желательно, чтобы все основные операции с множествами выполнялись достаточно быстро. Многие создатели трансляторов ограничивают максимальный размер множеств размером слова соответствующей машины. В этом случае каждый элемент множества представляется одним разрядом (0 означает отсутствие, а 1 — присутствие элемента в множестве.) Поэтому большинство реализаций не будет работать с множеством из 10 000 элементов. Это обстоятельство заставляет в программе 8.5 модифицировать представление данных.

Большое множество можно представить как массив меньших множеств, каждое из которых «влезает» в несколько слов (количество слов зависит от реализации). Новая программа основывается как на абстрактной модели уже на втором наброске. В ней и «решето» (Sieve), и простые числа (Primes) переопределяются как массивы множеств, а Next представляется записью.

```
program Prime3(Output);

   { Программа 8.5 — Поиск простых чисел в диапазоне 3 . . 10000
                     с помощью решета, содержащего нечетные числа
                                                  из этого диапазона. }
```

```
const
  SetSize = 128 { зависит от реализации };
  MaxElement = 127;
  Setparts = 39 { = 10000 div Setsize div 2 };

type
  Natural = 0..MaxInt;

var
  Sieve, Primes:
    array [0..SetParts] of
      set of 0..MaxElement;
  NextPrime:
    record
      Part, Element: Natural
    end;

    Multiple, NewPrime: Natural;
    P, N, Count: Natural;
    Empty: Boolean;

begin { инициация }
  for P := 0 to SetParts do
    begin Sieve[P] := [0 .. MaxElement]; Primes[P] := [] end;
  Sieve[0] := Sieve[0] - [0]; Empty := False;
  NextPrime.Part := 0; NextPrime.Element := 1;

  with NextPrime do
    repeat {поиск очередного простого }
      while not (Element in Sieve[Part]) do Element := Succ(Element);
      Primes[Part] := Primes[Part] + [Element];
      NewPrime := 2 * Element + 1;
      Multiple := Element; P := Part;
      while P <= SetParts do { eliminate }
        begin Sieve[P] := Sieve[P] - [Multiple];
          P := P + Part * 2;
          Multiple := Multiple + NewPrime;
          while Multiple > MaxElement do
            begin P := P + 1;
            Multiple := Multiple - SetSize
            end
```

```
      end;
   if Sieve[Part] = [] then
      begin  Empty := True;  Element := 0  end;
   while Empty and (Part < SetParts) do
      begin
         Part := Part + 1;  Empty := Sieve[Part] = []
      end
until Empty;

   Count := 0;
   for P := 0 to SetParts do
      for N := 0 to MaxElement do
         if N in Primes[P] then
            begin
               Write(Output, 2 * N + 1 + P * SetSize * 2:6);
               Count := Count + 1;
               if (Count mod 8) = 0 then Writeln(Output)
            end
end.
```

Дает в качестве результата:

```
     3     5     7    11    13    17    19    23
    29    31    37    41    43    47    53    59
    61    67    71    73    79    83    89    97
   101   103   107   109   113   127   131   137

     .     .                             .     .
     .                                   .     .
     .     .     .     .                 .     .

  9871  9883  9887  9901  9907  9923  9929  9931
  9941  9949  9967  9973 10007 10009 10037 10039
 10061 10067 10069 10079 10091 10093 10099 10103
 10111 10133 10139 10141 10151 10159 10163 10169
```

9

ФАЙЛОВЫЕ ТИПЫ

Во многих случаях наипростейшим методом введения структуры является последовательное расположение. Среди профессионалов по обработке данных в этой ситуации обычно используется термин *последовательный файл*. В Паскале слово *«файл»* относится к объектам, образованным последовательностью компонент, причем все они одного типа. Особые файлы, состоящие из строчек символов переменной длины, называются текстовыми файлами. Они лежат в основе всех механизмов взаимодействия человека с вычислительной системой.

9.1. СТРУКТУРА ФАЙЛА

Сама последовательность устанавливает естественный порядок компонент, причем в любой момент непосредственно доступна только одна компонента. Другие компоненты становятся доступными по мере продвижения по файлу. Число компонент, называемое *длиной* файла, в определении файлового типа не фиксируется. Эта особенность явно отличает понятие файла от понятия массива. Если файл не имеет компонент, его называют *пустым*. Таким образом, объекты файлового типа отличаются и от объектов массивового, и записного, и множественного типов тем, что они представляют собою структуру с последовательным доступом к компонентам одного типа.

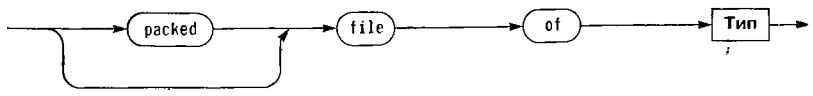


Рис. 9.1. Синтаксическая диаграмма для *Файлового типа*

Описание каждой переменной-файла F автоматически вводит некоторую *буферную переменную,* обозначаемую через F↑. Эта переменная относится к типу компонент файла. Буферную пере-

менную можно рассматривать как некоторое окно, через которое можно либо посмотреть (прочитать) на существующие компоненты, либо добавить (записать) новые компоненты. Окно автоматически передвигается при некоторых из операций над файлом. Всей файловой переменной присваивание делать нельзя. Вместо этого надо добавлять через буферную переменную по одной компоненте, т. е. действовать строго последовательным образом. Если файл находится в положении «за последней его компонентой», то значение буферной переменной не определено.

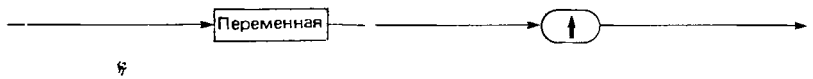


Рис. 9.2. Синтаксическая диаграмма для *Буферной переменной*

Последовательный характер обработки и наличие буферной переменной предполагают, что файлы могут ассоциироваться с внешней (вторичной) памятью и другим периферийным оборудованием. Как хранятся компоненты, зависит от реализации, мы лишь допускаем, что только некоторые из них находятся в какой-то момент в основной памяти, а непосредственно доступна только компонента, на которую указывает F↑.

Если буферная переменная F↑ дойдет до конца файла F, то предописанная логическая функция eof(F) станет давать значение true, в других случаях она дает значение false. Для работы с файлами предусмотрены такие процедуры:

Reset(F)	готовит файл для просмотра (чтения), помещая окно в начало файла. Если файл не пустой, то значение первой компоненты F присваивается F↑, а функция eof(F) начинает давать значение false
Rewrite(F)	готовит файл для формирования (записи). Значение F заменяется на пустой файл. Функция eof(F) начинает давать значение true, и можно записывать новый файл
Get(F)	передвигает окно файла на следующую компоненту и присваивает значение этой компоненты буферной переменной F↑. Если следующей компоненты нет, то eof(F) начинает давать значение true, а F↑ становится неопределенной. Если перед обращением к Get(F) значение eof(F) было true или файл находился в режиме формирования, то обращение считается ошибкой
Put(F)	добавляет значение буферной переменной F↑ к файлу. Обращение будет ошибкой, если перед ним предикат eof(F) не становится true. После добавления eof(F) остается true, а значение F↑ не определено. Если файл в режиме просмотра, то обращение — ошибка.

В принципе все действия по формированию и просмотру последовательных файлов можно полностью выразить с помощью этих четырех примитивных операций с файлами и предиката eof(F). Однако на практике естественнее объединять продвижение по файлу с обращением к буферной переменной. Поэтому мы вводим такие две процедуры: Read и Write.

Обращение Read(F, X), где X — переменная, эквивалентно:

```
begin
   X := F↑; Get(F)
end
```

Обращение Write(F, E), где E — выражение, эквивалентно:

```
begin
   F↑ := E; Put(F)
end
```

Фактически Read и Write в некотором роде специальные процедуры. Они приспособлены для работы с различным числом параметров (V1, ... Vn — переменные; E1, ..., En — выражения).

Обращение Read(F, V1, ..., Vn) эквивалентно оператору

begin Read(F, V1); ...;Read(F, Vn) end

а обращение Write(F, E1, ..., En) эквивалентно оператору

begin Write(F, E1); ...;Write(F, En) end

Преимущество использования таких процедур связано не только с их краткостью, но и с концептуальной простотой, так как мы можем игнорировать существование буфера F↑, значение которого бывает и неопределенным. Однако буфер порой бывает полезным для «заглядывания вперед».

Примеры описаний:

```
var Data: file of Integer;
    A: Integer;

var Plotfile: file of
                record
                   C: Color;
                   Len: Natural
                end;

var Club: file of Person;
    P: Person;
```

- *Примеры действий с файлами:*

```
A := Data↑; Get(Data)

Read(Data,A)

Plotfile↑ C := Red;
Plotfile↑.Len := 17;  Put(Plotfile)

Club↑ := P;  Put(Club)

Write(Club,P)
```

Файлы могут быть локальными по отношению к программе (или процедуре), однако они могут существовать и вне программы. Такие файлы называются внешними. Внешние файлы передаются как параметры в программу. Эти параметры перечисляются в заголовке программы (см. гл. 3).

Ниже приведены две программы, обрабатывающие файлы. Программа 9.1 имеет дело с файлом вещественных чисел, представляющих собой результаты измерений, выполненных каким-то инструментом либо другой программой. Программа же 9.2 работает с двумя файлами, где перечислены упорядоченные по «последнему» имени сотрудники:

F1, F2, ..., Fm и G1, G2, ..., Gn

причем $F(I+1) >= F(I)$ и $G(J+1) >= G(J)$ для всех I, J. Программа сливает эти два файла в один — H, так что

$$H(K+1) >= H(K) \text{ для } K = 1, 2, \ldots, (M+N-1).$$

```
program Normalize(DataIn, DataOut);

  { Программа 9.1 — Нормализация файла с изменениями:
                    вещественными числами, полученными из
                    внешнего источника. }

  type
    Measurements = file of Real;
    Natural = 0..MaxInt;

  var
    DataIn, DataOut: Measurements;
    Sum, Mean,
    SumOfSquares, StandardDeviation: Real;
    N: Natural;
begin
```

```
   Reset(DataIn);  N := 0;
   Sum := 0.0;  SumOfSquares := 0.0;

   while not eof(DataIn) do
      begin  N := N + 1;
         Sum := Sum + DataIn↑;
         SumOfSquares := SumOfSquares + Sqr(DataIn↑);
         Get(DataIn)
      end;
   Mean := Mean / N;
   StandardDeviation := Sqrt( (SumOfSquares / N) - Sqr(Mean) );
   Reset(DataIn);  Rewrite(DataOut);
   while not Eof(DataIn) do
      begin
         DataOut↑ := (DataIn↑ - Mean) / StandardDeviation;
         Put(DataOut);  Get(DataIn)
      end
end { Normalize }.

program MergeFiles(F,G,H);

   { Программа 9.2 — Слияние файлов F и G, отсортированных
                     по последним именам.}

   type
      Natural = 0..MaxInt;
      String15 = packed array [1 .15] of Char;
      Person =  record
                   Name:
                      record
                         First, Last: String15;
                      end;
                   Height: Natural {сантиметры } ;
                end;

   var
      F, G, H: file of Person;
      EndFG: Boolean;

begin
   Reset(F);  Reset(G);  Rewrite(H);
   EndFG := Eof(F) or Eof(G);
```

```
    while not EndFG do
      begin
        if F↑.Name.Last < G↑.Name.Last then
          begin  H↑ := F↑;  Get(F);  EndFG := Eof(F)
          end
        else
          begin  H↑ := G↑;  Get(G);  EndFG := Eof(G)
          end;
        Put(H)
      end;
    while not Eof(G) do
      begin
        Write(H, G↑);  Get(G)
      end;
    while not Eof(F) do
      begin
        Write(H, F↑);  Get(F)
      end
end
```

9.2. ТЕКСТОВЫЕ ФАЙЛЫ

Текстовыми называются файлы, состоящие из последовательности символов, разбитой на строки произвольной длины. Для описания таких файлов используется предопределенный тип Text.

Мы можем считать, что тип Text определен на базе типа Char, дополненного (гипотетическим) символом окончания строчки или маркером конца строки. Поэтому тип Text *не* эквивалентен (упакованному) файлу из символов. Маркер конца строки может распознаваться и формироваться такими специальными текстовыми процедурами:

Writeln(F)	заканчивает текущую строку (строчку) текстового файла F
Readln(F)	пропускает все до начала следующей строки текстового файла F (F↑ становится первым символом следующей строки)
Eoln(F)	логическая функция, указывающая, достигнут ли конец текущей строки в файле F. (Если true, то F↑ соответствует положению разделителя строк, на значение F↑ — пробел.)

Если F — текстовой файл, а Ch — символьная переменная, то можно пользоваться такими сокращениями:

Сокращенная форма	Полная форма
Write(F, Ch)	F↑ := Ch; Put(F)
Read(F, Ch)	Ch := F↑; Get(F)

Имена Input и Output выделены для двух стандартных переменных, представляющих собою текстовые файлы. Они используются в качестве параметров программы и обеспечивают чтение и запись текстов. Более детально о них пойдет речь в гл. 12, где будут описаны расширенные обращения к процедурам Read, Write, Readln и Writeln.

В приведенных ниже схемах программ мы воспользуемся упомянутыми соглашениями и продемонстрируем некоторые из типичных работ с файлами.

1. Формирование текстового файла Y. Предположим, с помощью P(C) вычисляется очередной символ и присваивается параметру C. Если нужно закончить текущую строчку, то логическая переменная B1 принимает значение true, если нужно закончить весь текст, то B2 — true.

```
Rewrite(Y);
repeat
   repeat P(C); Write(Y,C)
   until B1;
   Writeln(Y)
until B2
```

2. Чтение текстового файла X. Предположим, что с помощью Q(C) обрабатывается (очередной) символ C, а через R обозначено действие, которое надо выполнить, встретив конец строки.

```
Reset(X);
while not eof(X) do
   begin
      while not eoln(X) do
         begin Read(X,C); Q(C)
         end;
      R; Readln(X)
   end
```

3. Копирование текстового файла X в аналогичный файл Y с сохранением строчной структуры файла X.

```
Reset(X); Rewrite(Y);
While not eof(X) do
   begin { copy a line }
      while not eoln(X) do
         begin Read(X,C); Write(Y,C)
         end;
      Readln(X); Writeln(Y)
   end
```

Замечание о реализации. Кодирование (представление) маркера конца строки обычно связано с использованием управляющих символов. При символах множества ASCII для этого применяют два символа: cr (возврат каретки) и lf (перевод строки). Однако в некоторых системах используется множество символов, где таких символов нет. В этом случае для маркировки конца строк нужно воспользоваться другими приемами.

10

ССЫЛОЧНЫЕ ТИПЫ

Ранее речь шла о типах, предназначенных для описания статически размещенных переменных. *Статической переменной* называется переменная, описанная в программе и обозначаемая с помощью имени. Статической ее называют из-за того, что она существует, т. е. под нее выделена память на протяжении всего выполнения блока (программы, процедуры или функции), где она локализована. Однако переменные можно создавать и уничтожать динамически в процессе выполнения блока (без какой-либо связи со статической структурой программы). Такие переменные в дальнейшем будут называться *динамическими,* или *идентифицированными, переменными.*

10.1. ССЫЛОЧНЫЕ ПЕРЕМЕННЫЕ И ИДЕНТИФИЦИРОВАННЫЕ (ДИНАМИЧЕСКИЕ) ПЕРЕМЕННЫЕ

Идентифицированная (динамическая) переменная не указывается в описаниях переменных и ее нельзя обозначать с помощью имени. Вместо этого они порождаются и уничтожаются с помощью предописанных процедур New и Dispose, а идентифицируются они через ссылочные значения (представляющие собой не что иное, как адрес места в памяти, где находится вновь созданная переменная). Ссылочное значение должно быть присвоено уже существующей ссылочной переменной, относящейся к соответствующему ссылочному типу.

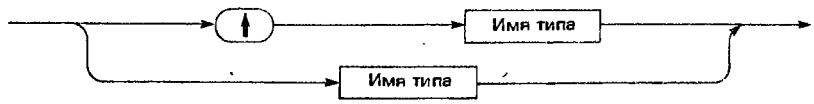


Рис. 10.1. Синтаксическая диаграмма для *Ссылочного типа*

Описание ссылочного типа P задает T — тип области:

type P = ↑T;

Множество ссылочных значений типа P представляет собой не-

ограниченное число *идентифицирующих значений,* каждое из которых идентифицирует переменную типа T, сюда же входит и специальное значение nil, не идентифицирующее никакую переменную.

Доступ к идентифицированной (динамической) переменной идет через идентифицирующее ее ссылочное значение. Если, например, переменная Ptr была описана так:

```
var Ptr: P;
```

и ей было присвоено некоторое идентифицирующее значение, то конструкция Ptr↑ обозначает идентифицированную переменную.

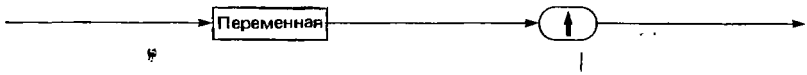


Рис. 10.2. Синтаксическая диаграмма для *Идентифицированной переменной*

Если Ptr равна nil или не определена, то обращение Ptr↑ — ошибочно.

Для порождения или размещения идентифицированной переменной типа T и присваивания идентифицирующего ее значения переменной Ptr используйте New(Ptr). А для уничтожения переменной, идентифицированной значением Ptr, применяйте Dispose(Ptr), после обращения к Dispose значение Ptr становится неопределенным.

Ссылки являются простым механизмом, позволяющим строить данные со сложной и меняющейся (и даже рекурсивной) структурой. Если тип T определяет записи с одним или несколькими полями типа P, то с их помощью можно строить структуры, эквивалентные произвольному конечному графу, причем идентифицированные переменные будут представлять вершины, а ссылки — ребра.

Программа 10.1 иллюстрирует, как с помощью ссылок можно моделировать очередь клиентов. (Процедуры рассматриваются в следующей главе.)

```
program WaitingList(Input,Output);

  { Программа 10.1 — Моделирование очереди клиентов;обслуживаются
                     первые 3. }
  const
    NameLength = 15;

  type
    NameIndex = 1..NameLength;
```

```
   NameString= packed array [NameIndex] of Char;
   Natural = 0..MaxInt;
   ClientPointer = ↑Client;
   Client =
      record
         Name: NameString;
         Nxt: ClientPointer
      end;

var
   Head, Tail: ClientPointer;
   Name: packed array [NameIndex] of Char;

procedure ReadName;
   var
      c: NameIndex;
begin
   for c := 1 to NameLength do
      if Eoln(Input) then Name[c] := ' '
      else
         begin  Read(Input,Name[c]);  Write(Output,Name[c])  end;
   Readln(Input);  Writeln(Output)
end { ReadName };

procedure AddClientToList;
   var
      NewClient: ClientPointer;
begin
   New(NewClient);
   if Head = nil then Head := NewClient
   else Tail↑.Nxt := NewClient;
   NewClient↑.Name := Name;  NewClient↑.Nxt := nil;
   Tail := NewClient
end { AddClientToList };

   procedure ServeClient(HowMany: Natural);
      var
         ClientToServe: ClientPointer;
   begin
      while (HowMany > 0) and (Head <> nil) do
         begin  ClientToServe := Head;  Head := Head↑.Nxt;
            Writeln(ClientToServe↑.Name); Dispose(ClientToServe);
            HowMany := HowMany - 1
         end
   end { ServeClients };

begin { WaitingList }
```

```
    Head := nil;
    while not Eof(Input) do
       begin  ReadName;  AddClientToList  end;
    Writeln(Output);
    ServeClients(3)
end { WaitingList }
```

Дает в качестве результатов:

```
Hikita
Balasubramanyam
Nagel
Lecarme
Bello
Pokrovsky
Barron
Yuen
Sale
Price

Hikita
Balasubramanyam
Nagel
```

В качестве другого примера рассмотрим «банк данных», включающий определенную группу лиц. Предположим, каждый человек представляется такой записью, которая была уже определена в гл. 7. Если добавить в нее поле ссылочного типа, то из таких записей можно формировать цепочки или связные списки и использовать их для поиска и включения информации:

```
type Link = ↑Person;
   ...
   Person = record
               ...
               Next: Link;
               ...
            end;
```

Связный список из n лиц можно представить так, как показано на рис. 10.3.

Каждый квадрат соответствует одному человеку:

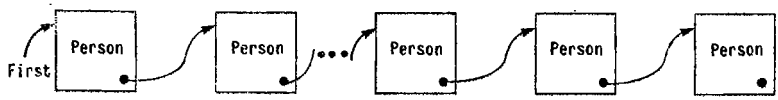


Рис. 10.3. Связный список

Переменная типа Link с именем First указывает на первого человека из этого списка. Поле же Next у «последнего человека» имеет значение nil. Заметим, что обозначение

First↑.Next↑.Next

указывает на третьего человека в списке.

Если допустить, например, что мы можем читать целые данные, относящиеся к росту человека, то с помощью приведенного ниже фрагмента программы можно построить упомянутую цепочку.

```
var First, P: Link; H,I: Integer;
        ...
First := nil
for I := 1 to N do
   begin Read(H); New(P);
      P↑.Next := First;
      P↑.Height := H; InitializeOtherFields(P↑);
      First := P
   end
```

Обратите внимание, что список растет в обратном направлении. Для обращения к элементам мы введем еще одну переменную, скажем Pt типа Link, которая будет свободно перемещаться по списку. Чтобы показать, как происходит поиск, представим себе, что есть информация о человеке с Height, равным 175, и нам нужно получить доступ к ней. В этом случае будем двигать Pt «вдоль» Next до тех пор, пока не обнаружим желаемое.

```
Pt := First;
while Pt↑.Height<> 175 do Pt := Pt↑.Next
```

Словами это можно выразить следующим образом: «Сначала Pt указывает на первого человека. До тех пор пока рост (Height) человека, на которого указывает Pt, не будет равен 175, будем присваивать Pt значение ссылки, находящееся в поле Next (это опять же ссылка) записи, на которую в данный момент указывает Pt».

Такая простая процедура поиска будет работать только в том случае, если есть уверенность, что найдется по крайней мере один человек с Height, равным 175. Но разве это реальное предположение? Поэтому для правильной работы нужно было бы еще и опознавать конец списка. Сначала это можно попытаться сделать так:

```
Pt := First;
while (Pt <> nil) and (Pt↑.Height <> 175) do
   Pt := Pt↑.Next
```

Вспомним, однако, о чем говорилось в разд. 4.1. Если Pt = nil, то переменная, на которую ссылаются во втором терме условия окончания, вообще не существует, это ошибка! Поэтому в данной ситуации можно выбрать любое из двух таких решений — и то и другое в этой ситуации работают правильно.

```
(1)  Pt:= First; B := true;
     while (Pt <> nil) and B do
     if Pt↑.Height = 175 then B := false else Pt := Pt↑.Next

(2)  Pt := First;
     while Pt <> nil do
        begin if Pt↑.Height = 175 then goto 13;
           Pt := Pt↑.Next
        end;
     13:
```

10.2. ФУНКЦИИ NEW и DISPOSE

Возьмем другую задачу: скажем, необходимо добавить в банк данных еще одно лицо. Для этого сначала с помощью предописанной процедуры New нужно выделить для переменной место и получить идентифицирующее ее значение:

New(P)	процедура размещает новую идентифицированную (динамическую) переменную P↑, относящуюся к области типа для P, и порождает новое идентифицирующее ссылочное значение, относящееся к тому же типу, что и P, и присваивает его P. Если P↑ — запись с вариантами, то New(P) выделяет место, достаточное для размещения любых вариантов
New(P, C1, ..., Cn)	размещает новую идентифицированную (динамическую) переменную P↑, относящуюся к вариантному записному типу для P, со значениями полей признаков C1, ..., Cn для n вложенных вариантов, и порождает новое идентифицирующее ссылочное значение, относящееся к тому же типу, что и P, и присваивает его P

Осторожно, если переменная-запись была порождена с помощью обращения к New второго типа, то во время выполнения программы нельзя переходить от одного варианта к другим. Присваивание всей целиком переменной считается ошибкой, хотя присваивания отдельным компонентам допускаются.

Начиная решать поставленную задачу, введем сначала ссылочную переменную; назовем ее NewP. В этом случае оператор

New(NewP)

разместит в памяти новую переменную типа Person. Затем новую переменную, на которую указывает ссылка NewP, нужно включить в список после элемента, на который ссылается Pt (рис. 10.4).

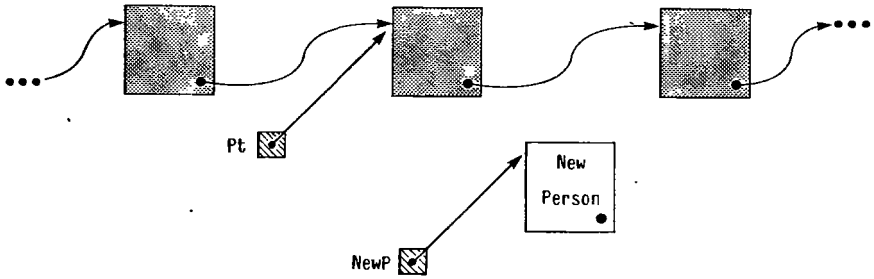


Рис. 10.4. Связный список перед включением

Включение проводится просто путем изменения ссылок.

```
NewP↑.Next := Pt↑.Next;
Pt↑.Next := NewP
```

Результат показан на рис. 10.5.

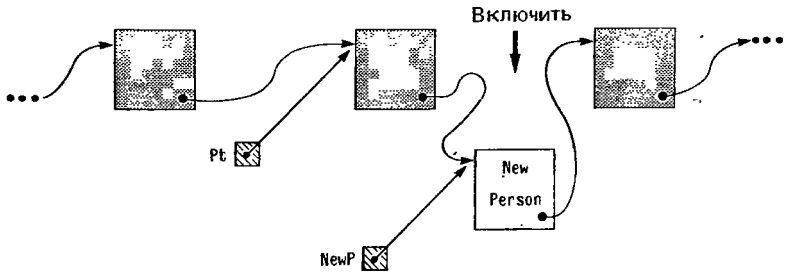


Рис. 10.5. Связный список после включения

Исключение элемента, следующего за тем, на который указывает вспомогательная переменная Pt, проводится в одно действие:

Pt↑.Next := Pt↑.Next↑.Next

При работе со списками часто бывает удобно пользоваться двумя ссылками, «смотрящими на соседние элементы». При исключении, например, хорошо, когда одна ссылка, скажем P1, указывает на элемент, предшествующий исключаемому, а вторая — P2 — на сам исключаемый. В этом случае исключение проводится за одно действие:

P1↑.Next := P2↑.Next

Однако существует опасность, что при таком исключении мы будем проигрывать в памяти (которую можно было бы еще использовать). Возможно в этом случае следует организовывать явный список «исключенных» элементов, на который указывает переменная Free. Тогда новые элементы можно брать из этого списка (если он не пуст), а не обращаться к процедуре New. При такой системе исключение из списка превращается в передачу элемента из исходного списка в список свободных элементов.

```
P1↑.Next := P2↑.Next;
P2↑.Next := Free;
Free := P2
```

И последнее, если воспользоваться предописанной процедурой Dispose, то работу с исключаемыми элементами можно возложить на реализацию Паскаля.

Dispose(Q)	убирает идентифицированную переменную Q↑ и уничтожает идентифицирующее значение Q. Если Q имело значение nil или было не определено, то — ошибка. Значение Q должно было быть порождено с помощью обращения к New первого вида
Dispose (Q, K1, ..., Kn)	убирает идентифицированную вариантную переменную-запись Q↑ с активными вариантами, селектированными значениями K1, ..., Kn. Соответствующее идентифицирующее значение Q уничтожается. Если Q имело значение nil или было не определено, то — ошибка. Значение Q должно было быть порождено с помощью обращения к New второго типа, а K1, ..., Kn должны иметь те же значения (выбирать те же варианты), что и при порождении

В гл. 11 приводятся две программы (программы 11.6 и 11.7), демонстрирующие обход древовидных образований, построенных с помощью ссылочных типов.

11

ПРОЦЕДУРЫ И ФУНКЦИИ

В процессе совершенствования искусства программирования мы стали создавать программы методом *последовательных уточнений*. На каждом этапе мы разбиваем задачу на несколько подзадач, определяя тем самым некоторое количество отдельных программ. Причем не стоит маскировать такую структуру программы. Концепция *процедуры* и *функции* как раз и позволяет нам выделять подзадачу как явную подпрограмму.

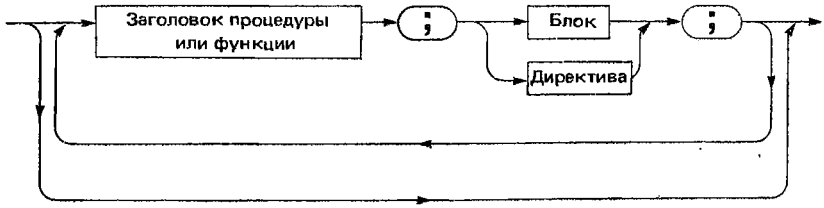


Рис. 11.1. Синтаксическая диаграмма
для *Раздела описания процедур и функций*

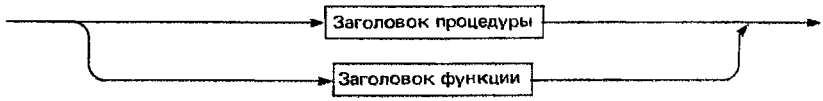


Рис. 11.2. Синтаксическая диаграмма
для *Заголовка процедуры или функции*

11.1. ПРОЦЕДУРЫ

На протяжении всей книги мы в примерах используем предописанные процедуры Read, Readln, Write и Writln. В этом же разделе объясняется, как программист сам может описывать процедуры. Фактически этот механизм уже был использован в программах 8.2 и 10.1.

Описание процедуры служит для определения части программы и сопоставления с ней некоторого имени, так что эту часть можно затем активировать с помощью *оператора процедуры*. Описание выглядит как программа, но вместо заголовка программы употребляется *заголовок процедуры*.

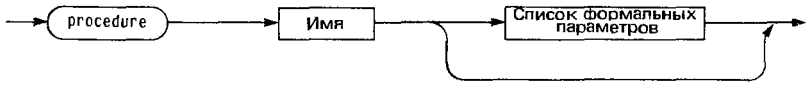


Рис. 11.3. Синтаксическая диаграмма для *Заголовка процедуры*

Вернемся к программе 6.1, с помощью которой находились минимальное и максимальное значения в списке целых чисел. Усложним ее; пусть к элементам A[1], ..., A[n] добавляются и приращений, а затем вновь отыскиваются Min и Max. В результате получим такую программу, где для определения Min и Max используется процедура.

```
program MinMax2(Input,Output);

  { Программа 11.1 — Введение в Программу 6.1 процедуры.  }

  const
    MaxSize = 20;

  type
    ListSize = 1..MaxSize;

  var
    Increment: Integer;
    Item: ListSize;
    A: array [ListSize] of Integer;

  procedure MinMax;
    var
      Item: ListSize;
      Min, Max, First, Second: Integer;
begin
  Min := A[1];  Max := Min;  Item := 2;
  while Item < MaxSize do
    begin  First := A[Item];  Second:= A[Item+1];
      if First > Second then
        begin
          if First > Max then Max := First;
          if Second < Min then Min := Second
        end
```

```
            else
               begin
                  if Second > Max then Max := Second;
                  if First < Min then Min := First
               end;
            Item := Item + 2
         end;
      if Item = MaxSize then
         if A[MaxSize] > Max then Max := A[MaxSize]
         else
            if A[MaxSize] < Min then Min := A[MaxSize];
      Writeln(Output, Max, Min); Writeln(Output)
   end {MinMax};

begin
   for Item := 1 to MaxSize do
      begin  Read(Input, A[Item]);  Write(Output, A[Item] :4)  end;
   Writeln(Output);
   MinMax;
   for Item := 1 to MaxSize do
      begin
         Read(Input, Increment);  A[Item] := A[Item] + Increment;
         Write(Output, A[Item] :4)
      end;
   Writeln(Output);
   MinMax
end
```

Дает в качестве результатов:

```
  -1  -3   4   7   8  54  23  -5   3   9   9   9  -6  45  79  79   3   1   1   5

        79        -6

  44  40   7  15   9  88  15  -4   7  43  12  17  -7  48  59  39   9   7   7  12

        88        -7
```

Хотя эта программа и очень проста, тем не менее из ее рассмотрения уже ясно, что:

1. Наипростейший *заголовок процедуры* выглядит так:

procedure *Имя*

2. *Блоки.* Любая процедура представляет собой поименованный блок. Имя блока программы — MinMax2, а блока процедуры — MinMax. В данном случае часть программы 6.1, отыскиваю-

щая минимальное и максимальное значения, изолируется и ей дается имя MinMax. Подобно блоку программы блок, образующий процедуру, имеет раздел описаний, в котором вводятся локальные по отношению к процедуре объекты.

3. *Локальные переменные.* В процедуре MinMax локальными являются переменные Item, First, Second, Min и Max; в программе (вне области действия MinMax) присваивание этим переменным не дает никакого результата. При каждой активации процедуры, перед началом выполнения ее раздела операторов, локальные переменные не определены.

4. *Глобальные переменные.* Переменные A, Item, Increment — глобальные переменные, описанные в главной программе. На них можно ссылаться по всей программе (например, первое в MinMax присваивание — Min := A[1]).

5. *Область действия.* Обратите внимание, что под именем Item фигурирует и локальная, и глобальная переменная. Это не одна и та же переменная! Из процедуры можно ссылаться на любую нелокальную переменную, однако такое имя можно и переописать. Если имя некоторой переменной переопределено, то в области действия переопределяющей процедуры имеет силу уже новая связь между именем и типом, и глобальная переменная с таким именем в этой области действия уже недоступна (если только она не передается как параметр). Присваивание локальной переменной Item (например, Item := Item + 2) не оказывает никакого влияния на глобальную переменную Item, и поскольку в MinMax предпочтение отдается именно локальной Item, то к глобальной Item обратиться фактически невозможно.

Практика хорошего программирования говорит, что если на имя нет ссылок вне процедуры, то его следует описывать как строго локальное в этой процедуре. Это требование не только хорошей документируемости, но и дополнительной безопасности. Например, Item можно было бы оставить и глобальной переменной, но тогда последующее расширение программы, где к процедуре MinMax обращаются из цикла с параметром Item, приведет к нарушению ее нормального выполнения.

6. *Оператор процедуры.* Оператор MinMax в главной программе активирует работу процедуры*. Такие операторы называются операторами процедуры.

---

* Ничто не отличает его от имени процедуры. — *Примеч. пер.*

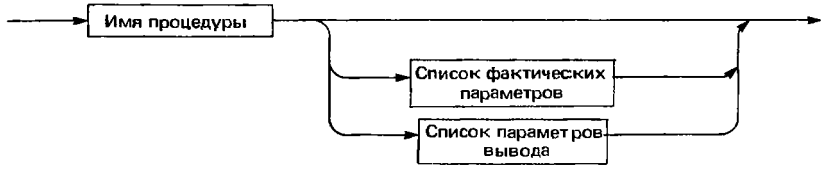


Рис. 11.4. Синтаксическая диаграмма для *Оператора процедуры*

Внимательно разбирая программу 11.1, можно заметить, что MinMax активируется дважды. Оформляя фрагмент программы как процедуру, а не выписывая явно этот фрагмент два раза, программист не только экономит на времени написания программы, но и на памяти, занимаемой его программой. Статический фрагмент программы на языке машины хранится только в одном месте, а память под локальные переменные динамически выделяется в процессе выполнения процедуры (при входе в нее память захватывается, а при выходе — освобождается).

Не следует, однако, колебаться, оформлять или нет некоторое действие как процедуру (даже когда к ней обращаются лишь единожды), если это приводит к улучшению читаемости программы. Как правило, в коротких блоках легче разбираться, чем в длинных. При выделении этапов разработки как процедур программу будет легче передать другому человеку и легче ее проверить.

11.1.1. Списки параметров

При разбиении задачи на подпрограммы часто бывает необходимо вводить новые переменные для аргументов и результатов этих подпрограмм. Назначение таких переменных должно быть ясно из текста программы.

Ниже приводится программа 11.2, представляющая собой очередное обобщение примера определения минимального и максимального значений среди элементов массива. Она позволяет познакомиться с такими особенностями процедур:

1. *Заголовок процедуры* бывает и с параметрами, т. е. второго вида.

2. *Формальные параметры.* В списке параметров даются имена каждому из параметров, а затем указывается их тип. Процедура MinMax имеет следующие параметры: L, Min и Max. Список формальных параметров открывает новую область действия для параметров.

3. *Фактические параметры.* Обратите внимание на соответствие между заголовком процедуры и оператором процедуры. По-

следний содержит список *фактических параметров,* которые подставляются вместо соответствующих формальных параметров, определяемых в описании процедуры. Соответствие устанавливается на основе положения параметров в том и другом списке. Параметры обеспечивают некоторый механизм подстановки, позволяющий повторять процесс, изменяя его аргументы (например, MinMax вызывается два раза: один раз для просмотра массива A, другой раз — для B).

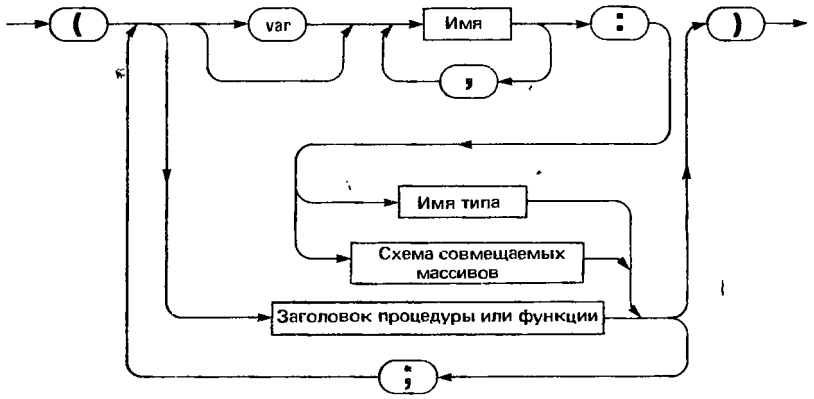


Рис. 11.5. Синтаксическая диаграмма для *Списка формальных параметров*

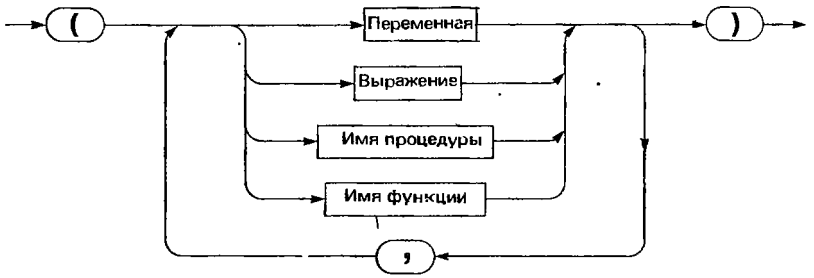


Рис. 11.6. Синтаксическая диаграмма для *Списка фактических параметров*

Параметры бывают четырех видов: параметры-значения, параметры-переменные, процедуральные параметры (они описываются в разд. 11.1.4) и функциональные параметры (о них см. в разд. 11.2.1).

```
program MinMax3(Input,Output);

  {Программа 11.2 – Модификация Программы 11.1 для двух списков.}

  const
    MaxSize = 20;

  type
    ListSize = 1..MaxSize;
    List = array [ListSize] of Integer;
  var
    Item: ListSize;
    A, B: List;
    MinA, MinB, MaxA, MaxB: Integer;

  procedure MinMax(var L: List; var Min, Max: Integer);
    var
      Item: ListSize;
      First, Second: Integer;

  begin
    Min := L[1];  Max := Min;  Item := 2;
    while Item < MaxSize do
      begin  First := L[Item];  Second:= L[Item+1];
        if First > Second then
          begin
            if First > Max then Max := First;
            if Second < Min then Min := Second
          end
        else
          begin
            if Second > Max then Max := Second;
            if First < Min then Min := First
          end;
        Item := Item + 2
      end;
    if Item = MaxSize then
      if L[MaxSize] > Max then Max := L[MaxSize]
      else
        if L[MaxSize] < Min then Min := L[MaxSize]
  end { MinMax };

  procedure ReadWrite(var L: List);
  begin
    for Item := 1 to MaxSize do
```

```
        begin  Read(Input, L[item]);  Write(Output, L[Item] :4)  end;
      Writeln(Output)
    end { ReadWrite };

begin  { main program }
   ReadWrite(A);
   MinMax(A, MinA, MaxA);
   Writeln(Output, MinA, MaxA, MaxA - MinA);  Writeln(Output);
   ReadWrite(B);
   MinMax(B, MinB, MaxB);
   Writeln(Output, MinB, MaxB, MaxB - MinB);  Writeln(Output);
   Writeln(Output);
   Writeln(Output, abs(MinA - MinB), abs(MaxA - MaxB));
   Writeln(Output);
   for Item := 1 to MaxSize do
      begin
         A[Item] := A[Item] + B[Item];
         Write(Output, A[Item] :4)
      end;
   Writeln(Output);
   MinMax(A, MinA, MaxA);
   Writeln(Output, MinA, MaxA, MaxA - MinA)
end
```

Дает в качестве результата:

```
 -1  -3   4   7   8  54  23  -5   3   9   9   9  -6  45  79  79   3   1   1   5
       -6          79          85

 45  43   3   8   1  34  -8   1   4  34   3   8  -1   3  -2  -4   6   6   6   7
       -8          45          53


        2          34

 44  40   7  15   9  88  15  -4   7  43  12  17  -7  48  77  75   9   7   7  12
       -7          88          95
```

4. *Параметры-переменные.* В программе MinMax мы имеем дело с *параметрами-переменными.* В этом случае фактический параметр должен быть переменной, а перед соответствующим формальным параметром должно стоять служебное слово var, после этого на все время выполнения процедуры формальный параметр становится синонимом этой фактической переменной. Любые операции с таким формальным параметром выполняются непосредственно над соответствующим фактическим параметром. Параметры-пере-

менные следует использовать для представления *результатов* процедур, как, например, используются Min и Max в процедуре MinMax. Более того, если X1, ..., Xn — фактические переменные, соответствующие формальным параметрам-переменным V1, ..., Vn, то все X1, ..., Xn должны быть *разными* переменными. Вычисление всех адресов производится в момент активации процедуры. Следовательно, если переменная есть компонента массива, то ее индексное выражение вычисляется при обращении к процедуре. Заметим, что ни компонента упакованной структуры, ни поле признака из записи с вариантами не должны фигурировать в качестве фактических параметров-переменных, чтобы не возникали зависящие от реализации проблемы вычисления адресов.

Если в начале раздела параметров никакого служебного слова нет, то речь идет о *параметрах-значениях*. В этом случае фактические параметры *должны быть выражениями* (в наипростейшем случае это переменная). Соответствующий формальный параметр представляется в вызываемой процедуре некоторой локальной переменной. В качестве начального значения этой переменной берется текущее значение соответствующего фактического параметра (т. е. значение выражения в момент активации). После этого процедура может изменять значение такой переменной, например, с помощью присваивания. На значение фактического параметра такое присваивание не окажет никакого влияния. Следовательно, параметр-значение никак нельзя использовать в качестве результата вычислений. Обратите внимание, что параметры-файлы или составные переменные с компонентами-файлами нельзя специфицировать как параметры-значения, поскольку они подразумевают присваивания.

Разница между параметрами-переменными и параметрами-значениями хорошо видна на примере программы 11.3.

```
program Parameters(Output);

{ Программа 11.3 – Пример параметров-значений и параметров-переменных.}

  var
    A, B: Integer;

  procedure Add1(X: Integer; var Y: Integer);
  begin
    X := X + 1;  Y := Y + 1;  Writeln(Output,X,Y)
  end { Add1 };

begin
  A := 0;  B := 0;  Add1(A,B);
```

```
    Writeln(Output,A,B)
end { Parameters }.
```

Дает в качестве результатов:

```
1       1
0       1
```

В приведенной ниже таблице суммируются сведения о соответствии между списком фактических и формальных параметров.

	Формальный параметр	Фактический параметр
параметр-значение	имя переменной	выражение
параметр-переменная	имя переменной	переменная
процедуральный параметр	заголовок процедуры	имя процедуры
функциональный параметр	заголовок функции	имя функции

В процедуре MinMax из программы 11.2 ни одно из значений в массиве L не изменяется, т. е. L — не результат. Следовательно, L можно было бы определить как параметр-значение, не влияя при этом на конечный результат. Однако чтобы понять, почему это не делается, полезно разобраться в реализации.

При любой *активации* процедуры для каждого параметра-значения выделяется новое место в памяти. Текущее значение фактического параметра копируется туда, а при выходе из процедуры эта память просто освобождается.

Если параметр не используется для передачи результатов процедуры, то предпочтительнее пользоваться параметром-значением. Обращение к нему идет быстрее, и, кроме того, есть защита от ошибочного изменения данных. Однако в случае параметра сложного типа (например, массива) нужно соблюдать осторожность, поскольку операция копирования относительно дорога и для копии может потребоваться слишком много места в памяти. Так как мы обращаемся к каждому элементу массива L лишь по одному разу, то лучше описать соответствующий параметр как параметр-переменную.

Размер массива изменяется путем простого переопределения MaxSize. Если программу необходимо применять к массивам вещественных чисел, то нужно изменить только определение типа и переменных; на операторах же никак не сказывается, что данные целые.

11.1.2. Совмещаемые массивы-параметры

Для передачи в процедуру или функцию массивов переменного размера можно воспользоваться и другим способом: использовать в качестве параметров-значений или параметров-переменных совмещаемые массивы-параметры. Однако будьте осторожны, в стандартном Паскале такая возможность лишь допускается и некоторые реализации ее не предусматривают.

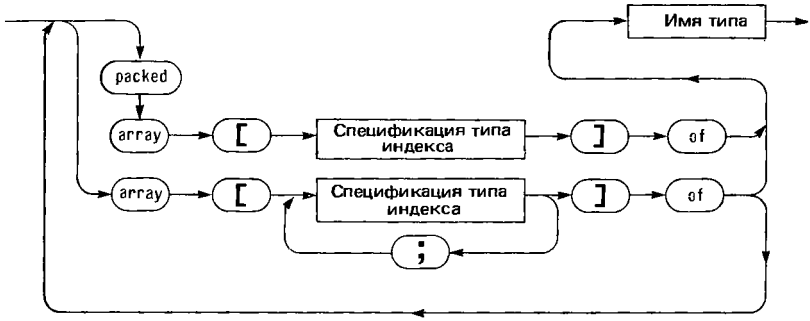


Рис. 11.7. Синтаксическая диаграмма для *Схемы совмещаемого массива*

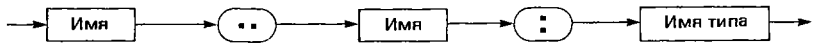


Рис. 11.8. Синтаксическая диаграмма для *Спецификации типа индекса*

Совмещаемые массивы задают реальные границы по каждой размерности через имена границ, представляющие собой в некотором роде информацию, которую можно лишь читать. Тип индекса фактического массива-параметра должен быть совместим с типом из спецификации типа индекса для совмещаемого массива. Минимальное и максимальное значения этого типа индекса должны лежать внутри замкнутого интервала типа индекса из спецификации. Тип компонент должен быть одинаковым, и если тип компонент совмещаемого массива-параметра в свою очередь совмещаемый массив-параметр, то тип компонент фактического массива-параметра должен быть совместим с ним.

Любой совмещаемый массив-параметр может быть упакован лишь по последней размерности. Фактическими параметрами для совмещаемых массивов-параметров-значений могут быть переменные или ст оки.

Программа MatrixMul из гл. 6 при использовании совмещаемых массивов-параметров может при переписывании превратиться в программу 11.4. В программе же 11.7 демонстрируется передача через формальный совмещаемый массив-параметр строк различного размера.

11.1.3. Рекурсивные процедуры

Использование имени процедуры в тексте самой процедуры предполагает *рекурсивное* выполнение этой процедуры. Задачи, естественным образом формулируемые как рекурсивные, часто приводят к рекурсивным решениям. В качестве примера рассмотрим программу 11.5.

Проблема заключается в том, что нужно построить программу, преобразующую выражения в постфиксную нотацию («польскую запись»). Это делается с помощью преобразующих процедур для каждой из синтаксических конструкций (выражения, слагаемого, множителя). Поскольку эти синтаксические конструкции определяются рекурсивно, то соответствующие процедуры могут рекурсивно активировать сами себя.

В качестве исходных данных программа должна воспринимать такие, например, символьные выражения:

```
(a+b)*(c-d)
a+b*c-d
(  a    * b)*  c-d
a+b*(c-d)
a*a*a*a
b+c*(d+c*a*a)*b+a
```

Выражения строятся в соответствии со следующими формулами РБНФ:

Выражение = *Слагаемое* {(«+»|«—») *Слагаемое*}.
Слагаемое = *Множитель* {«*» *Множитель*}.
Множитель = *Имя*|«(«*Выражение*»)».
Имя = *Буква*.

```
program MatrixMul2(Input, Output);
```

{Программа 11.4 — Вариант Программы 6.3, процедура использует совмещаемые массивы-параметры.}

```
const
   M = 4;  P = 3;  N = 2;

type
   Positive = 1..MaxInt;

var
   A: array [1..M, 1..P] of Integer;
   B: array [1..P, 1..N] of Integer;
   C: array [1..M, 1..N] of Integer;

procedure ReadMatrix
   (var X: array [LoRow..HiRow: Positive;
                  LoCol..HiCol: Positive] of Integer);
   var
      Row, Col: Positive;
begin
   for Row := 1 to HiRow do
      for Col := 1 to HiCol do
         Read(Input, X[Row,Col])
end { ReadMatrix };

procedure WriteMatrix
   (var X: array [LoRow..HiRow: Positive;
                  LoCol..HiCol: Positive] of Integer);
   var
      Row, Col: Positive;
begin
   for Row := 1 to HiRow do
      begin
         for Col := 1 to HiCol do
            Write(Output, X[Row,Col]);
         Writeln(Output)
      end
end { WriteMatrix }

procedure Multiply
   (var A: array [LoARow..HiARow: Positive;
                  LoACol..HiACol: Positive] of Integer;
    var B: array [LoBRow..HiBRow: Positive;
                  LoBCol..HiBCol: Positive] of Integer;
    var C: array [LoCRow..HiCRow: Positive;
                  LoCCol..HiCCol: Positive] of Integer);
```

```
      var
         Sum: Integer;
         I, J, K: Positive;
   begin
      if (LoARow <> 1) or (LoACol <> 1) or (LoBRow <> 1) or
         (LoBCol <> 1) or (LoCRow <> 1) or (LoCCol <> 1) or
         (HiARow <> HiCRow) or (HiACol <> HiBRow) or
         (HiBCol <> HiCCol) then {error}
      else
         for I := 1 to HiCRow do
            begin
               for J := 1 to HiCCol do
                  begin  Sum := 0;
                     for K := 1 to HiACol do
                        Sum := Sum + A[I,K] * B[K,J];
                     C[I,J] := Sum
                  end;
            end
   end { Multiply };

begin
   ReadMatrix(A);
   WriteMatrix(A);
   ReadMatrix(B);
   WriteMatrix(B);
   Multiply(A,B,C);
   WriteMatrix(C)
end
```

Дает в качестве результатов:

```
 1   2   3
-2   0   2
 1   0   1
-1   2  -3

-1   3
-2   2
 2   1

 1  10
 6  -4
 1   4
-9  -2
```

```
program PostFix(Input,Output);

  { Программа 11.5 — Преобразование инфиксных выражений
                     в постфиксную нотацию }
  label 13 {реакция на конец файла};

  var
     Ch: Char;
  procedure Find;
  begin
     if Eof(Input) then goto 13;
     repeat  Read(Input, Ch);
     until (Ch <> ' ') or Eof(Input)
  end { Find };

  procedure Expression;
     var
        Op: Char;
     procedure Term;

        procedure Factor;
        begin
           if Ch = '(' then
              begin  Find;  Expression;  { Ch = ')' }  end
           else
              Write(Output, Ch);
           Find
        end { Factor };

     begin { Term }
        Factor;
        while Ch = '*' do
           begin  Find;  Factor;  Write(Output, '*')
           end
     end { Term };

  begin { Expression }
     Term;
     while (Ch = '+') or (Ch = '-') do
        begin
           Op := Ch;  Find;  Term;  Write(Output, Op)
        end
  end { Expression };
```

```
begin { PostFix }
   Find;
   repeat
      Expression;
      Writeln(Output)
   until Ch = '.';
13:
end { PostFix } .
```

Дает в качестве результатов:

```
ab+cd-*
abc*+d-
ab+c*d-
abcd-*+
aa*a*a*
bcdca*a*+*b*+a+
```

Двоичное дерево — это такая структура данных, которая естественно определяется с помощью рекурсии и обрабатывается рекурсивными алгоритмами. Она состоит из конечного множества вершин, которое либо пусто, либо содержит вершину (корневую) с двумя присоединенными к ней двоичными деревьями, называемыми левым и правым поддеревьями [6]. Рекурсивные процедуры для формирования и обхода двоичных деревьев естественным образом отражают рекурсивную природу такого определения.

Программа 11.6 строит двоичное дерево и обходит его в прямом, естественном и обратных порядках. Само дерево задается в прямом порядке, т. е. путем перечисления вершин (в данном случае обозначаемых одной буквой), начиная с корня, затем сначала левого поддерева, а потом правого. Например, дерево, приведенное на рис. 11.9, вводится как последовательность

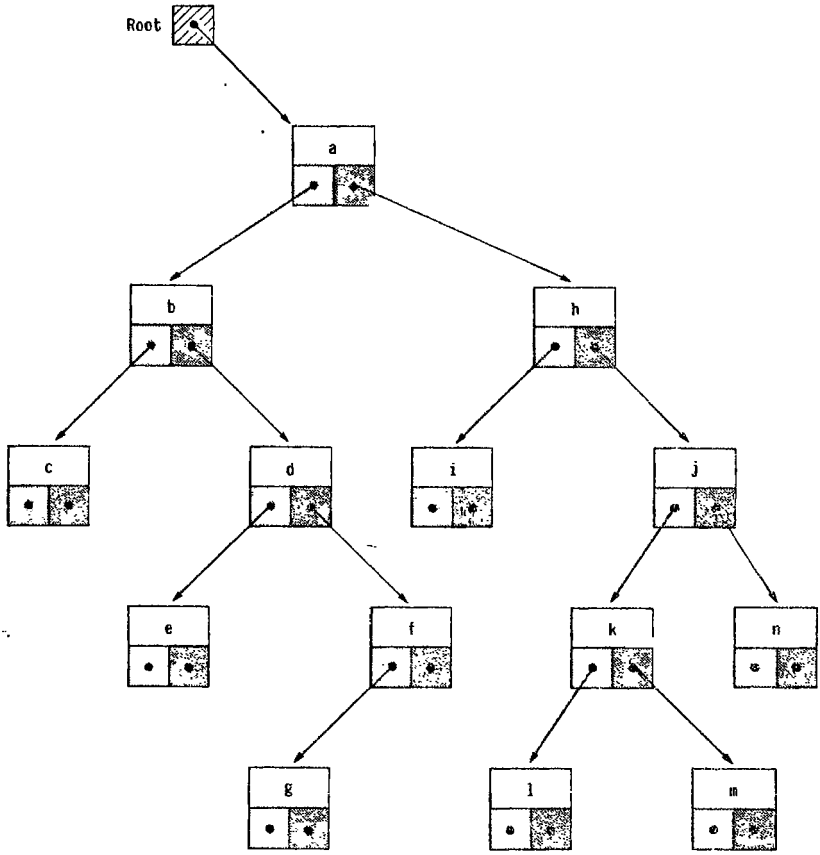
abc..de..fg...hi..jkl..m..n..

причем точками обозначены пустые поддеревья.

11.1.4. Процедуральные параметры

Для иллюстрации возможности передачи процедуры в качестве параметра мы перепишем программу 11.6. В списке формальных параметров процедуры или функции процедуральные параметры выглядят как заголовки процедур. В соответствующем же месте списка фактических параметров должно обязательно указываться

имя процедуры. Кроме того, в программе 11.7 показано, как фактическое значение строки передается на место совмещаемого массива-параметра.



Р и с. 11.9. Строение двоичного дерева

```
program Traversal(Input,Output);

  { Программа 11.6 – Обход двоичного дерева }

  type
    Ptr = ↑Node;
```

```
   Node =
      record
         Info: Char;
         LLink, RLink: Ptr
      end;

   var
      Root: Ptr;
      Ch: Char;
   procedure PreOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            Write(Output,P↑.Info); PreOrder(P↑.LLink); PreOrder(P↑.RLink)
         end
   end { PreOrder };
   procedure InOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            InOrder(P↑.LLink); Write(Output, P↑.Info); InOrder(P↑.RLink)
         end
   end { InOrder };
   procedure PostOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            PostOrder(P↑.LLink);PostOrder(P↑.RLink);Write(Output,P↑.Info)
         end
   end { PostOrder };
   procedure Enter(var P: Ptr);
   begin  Read(Input, Ch);  Write(Output, Ch);
      if Ch <> '.' then
         begin  New(P);
            P↑.Info := Ch; Enter(P↑.LLink); Enter(P↑.RLink)
         end
      else  P := nil
   end { Enter };
begin { Traversal }
   Enter(Root);  Writeln(Output);
   PreOrder(Root);  Writeln(Output);
   InOrder(Root);  Writeln(Output);
   PostOrder(Root);  Writeln(Output)
end { Traversal }
```

Дает в качестве результатов:

```
abc..de..fg...hi..jkl..m..n..
abcdefghijklmn
cbedgfaihlkmjn
cegfdbilmknjha
```

```
program Traversal2(Input,Output);

  { Программа 11.7 — Вариант Программы 11.6 с параметрами-процедурами. }
   type
      Ptr = ↑Node;
      Node =
         record
            Info: Char;
            LLink, RLink: Ptr
         end;
      Positive = 1..MaxInt;
   var
      Root: Ptr;
      Ch: Char;
   procedure PreOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            Write(Output,P↑.Info); PreOrder(P↑.LLink); PreOrder(P↑.RLink)
         end
   end { PreOrder };

   procedure InOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            InOrder(P↑.LLink); Write(Output, P↑.Info); InOrder(P↑.RLink)
         end
   end { InOrder };

   procedure PostOrder(P: Ptr);
   begin
      if P <> nil then
         begin
            PostOrder(P↑.LLink); PostOrder(P↑.RLink); Write(Output,P↑.Info)
         end
   end { PostOrder };
```

```
  procedure Enter(var P: Ptr);
  begin  Read(Input, Ch); Write(Output, Ch);
    if Ch <> '.' then
      begin  New(P);
        P↑.Info := Ch; Enter(P↑.LLink); Enter(P↑.RLink)
      end
    else  P := nil
  end { Enter };

  procedure WriteNodes(procedure TreeOperation(Start: Ptr); Root: Ptr;
                       Title: packed array [M..N: Positive] of Char);
    var
      C: Positive;
  begin
    Writeln(Output);
    for C := M to N do Write(Output, Title[C]);
    Writeln(Output); Writeln(Output);
    TreeOperation(Root); Writeln(Output)
  end { WriteNodes };

begin { Traversal2 }
  Enter(Root); Writeln(Output);
  WriteNodes(PreOrder, Root, 'Nodes listed in preorder:');
  WriteNodes(InOrder, Root, 'Nodes listed inorder:');
  WriteNodes(PostOrder, Root, 'Nodes listed in postorder:')
end { Traversal2 }.
```

Дает в качестве результатов:

```
abc..de..fg. .hi..jkl..m..n..

Nodes listed in preorder:

abcdefghijklmn

Nodes listed inorder:

cbedgfaihlkmjn

Nodes listed in postorder:

cegfdbilmknjha
```

Следует предостеречь читателя от использования рекурсивных методов без должного анализа проблемы. Хотя они и выглядят иногда «прозрачными», но не всегда приводят к самым эффективным решениям.

Если процедура P активирует Q, а Q в свою очередь активирует P, то либо P, либо Q должны быть заранее упомянуты с помощью *опережающего описания* (см. разд. 11.3).

В каждой реализации стандартного Паскаля должны быть предусмотрены предописанные *процедуры*, приводящиеся в приложении 1. В зависимости от реализации сюда могут добавляться и другие подобные процедуры. Аналогично всем предописанным или предопределенным объектам считается, что они находятся в области действия, окружающей программу пользователя. Поэтому при переописании соответствующего имени внутри программы никаких неприятностей не будет.

Предописанные процедуры нельзя передавать в качестве процедуральных фактических параметров.

11.2. ФУНКЦИИ

Функция — это часть программы (в некотором смысле аналогичная процедуре), определяющая одно-единственное ординальное, вещественное или ссылочное значение, используемое при вычислении выражения. Активация функции вызывается написанием *обозначения функции*, состоящим из имени обозначаемой функции и некоторого списка фактических параметров. Параметрами могут быть переменные, выражения, процедуры или функции; они подставляются вместо соответствующих формальных параметров.

Описание функции такое же, как и программы, но только *заголовок функции* имеет вид:

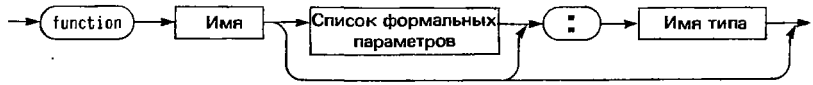


Рис. 11.10. Синтаксическая диаграмма для *Заголовка функции*

Как и в случае процедуры, метки из раздела меток и все имена, введенные в разделах определения констант и типов, а также в разделах описания переменных, процедур или функций, являются локальными по отношению к описанию функции, которое и называется областью действия для этих объектов. Вне области действия они неизвестны. В начале раздела операторов значения локальных переменных не определены.

Имя, указанное в заголовке функции, именует эту функцию. Тип результата должен быть простым или ссылочным. Внутри описания функции должно выполняться присваивание (с типом результата) имени функции. Такое присваивание «возвращает» результат функции.

В программе 11.8 алгоритм возведения в степень из программы 4.3 оформляется как описание функции.

Если имя функции появляется где-либо в выражении внутри самой функции, то речь идет о рекурсивном выполнении функции. Первый из примеров приложения 6 иллюстрирует использование рекурсивной функции.

Обозначение функции может встречаться раньше ее определения, если есть *опережающее описание* (см. разд. 11.3).

Предполагается, что в каждой реализации стандартного Паскаля предопределены *функции*, перечисляемые в приложении 1. При реализации их состав может дополняться. Предописанные функции нельзя передавать как функциональные фактические параметры.

```
program Exponentiation2(Output);

  { Программа 11.8 — Вариант Программы 4.6 с функцией. }

  type
    Natural = 0..MaxInt;

  var
    Pi, PiSquared: Real;

  function Power(Base: Real; Exponent: Natural): Real;
    var
      Result: Real;
  begin Result := 1;
    while Exponent > 0 do
      begin
        while not Odd(Exponent) do
          begin Exponent := Exponent div 2; Base := Sqr(Base)
          end;
        Exponent := Exponent - 1; Result := Result * Base
      end;
    Power := Result
  end { Power };

begin Pi := ArcTan(1.0) * 4;
  Writeln(Output, 2.0 :11:6, 7 :3, Power(2.0,7) :11:6);
  PiSquared := Power(Pi,2);
  Writeln(Output, Pi  :11:6, 2 :3, PiSquared    :11:6);
  Writeln(Output, PiSquared :11:6, 2 :3, Power(PiSquared, 2) :11:6);
  Writeln(Output, Pi  :11:6, 4 :3, Power(Pi,4)  :11:6)
end { Exponentiation2 } .
```

Дает в качестве результатов:

```
2.000000  7 128.000000
3.141593  2   9.869605
9.869605  2  97.409100
3.141593  4  97.409100
```

11.2.1. Функциональные параметры

Функции сами могут передаваться в качестве параметра другим функциям или процедурам. Формальный функциональный параметр специфицируется с помощью заголовка функции, соответствующий ему фактический параметр должен быть именем функции. Программа 11.9 определяет суммы элементов ряда для функции, задаваемой в момент обращения.

```
program SumSeries(Output);

  { Программа 11.9 — Печать таблицы сумм рядов. }

  const
    MaxTerms = 10;

  var
    Term: 1..MaxTerms;

  function Sigma(function F(X:Real):Real;Lower,Upper:Integer):Real;
    var
      Index: Integer;
      Sum: Real;
  begin
    Sum := 0.0;
    for Index := Lower to Upper do
      Sum := Sum + F(Index);
    Sigma := Sum
  end { Sigma };

  function IncreasingSine(X: Real): Real;
  begin
    IncreasingSine := sin(X) * X
  end { IncreasingSine };

  function InverseCube(X: Real): Real;
  begin
    InverseCube := 1 / (Sqr(X) * X)
  end { InverseCube };
```

```
begin { SumSeries }
   for Term := 1 to MaxTerms do
     Writeln(Term,Sigma(IncreasingSine,1,Term), Sigma(InverseCube,1,Term))
end { SumSeries } .
```

Дает в качестве результата:

```
 1 8.414710E-01 1.000000E+00
 2 2.660066E+00 1.125000E+00
 3 3.083426E+00 1.162037E+00
 4 5.621672E-02 1.177662E+00
 5-4.738405E+00 1.185662E+00
 6-6.414900E+00 1.190292E+00
 7-1.815995E+00 1.193207E+00
 8 6.098872E+00 1.195160E+00
 9 9.807942E+00 1.196532E+00
10 4.367733E+00 1.197532E+00
```

11.2.2. Побочный эффект

Встречающееся внутри описания функции присваивание нелокальной переменной или параметру-переменной называется *побочным эффектом.* Часто это лишь затуманивает назначение программы и значительно усложняет ее верификацию. Поэтому использование функций, дающих побочный эффект, вряд ли стоит рекомендовать. В качестве примера разберите программу 11.10.

11.3. ОПЕРЕЖАЮЩЕЕ ОПИСАНИЕ

Имя процедуры (или функции) можно использовать до описания процедуры (или функции), если есть *опережающее описание.* Опережающее описание необходимо в случае взаиморекурсивных процедур и не вложенных одна в другую функций. Такое описание выглядит следующим образом (обратите внимание, что список параметров и тип результата записываются только в опережающем упоминании):

```
procedure Q(X: T); Forward;

procedure P(Y: T);
begin
   Q(A)
end;
```

```
procedure Q; { параметры и тип результата не повторяются }
begin
   P(B)
end;
program SideEffect(Output);

   {Программа 11.10 — Пример побочного эффекта функции.}

   var
      A, Z: Integer;

   function Sneaky(X: Integer): Integer;
   begin
      Z := Z - X { side effect on Z };
      Sneaky := Sqr(X)
   end { Sneaky };

begin
   Z := 10;  A := Sneaky(Z);
   Writeln(Output, A, Z);
   Z := 10;  A := Sneaky(10);  A := A * Sneaky(Z);
   Writeln(Output, A, Z);
   Z := 10;  A := Sneaky(Z);  A := A * Sneaky(10);
   Writeln(Output, A, Z);
end { SideEffect }
```

Дает в качестве результата:

```
      100         0
        0         0
    10000       -10
```

12

ТЕКСТОВЫЕ ФАЙЛЫ. ВВОД И ВЫВОД

В гл. 9, говоря о текстовых файлах, мы уже упоминали о проблеме взаимодействия человека и машины. И тот и другой учатся понимать партнера на основании процесса, получившего название «опознание образов». К несчастью, образы, опознание которых легче всего выполняется человеком (зрительные или слуховые), весьма отличаются от тех, которые воспринимает машина (электрические импульсы). Фактически стоимость физической передачи данных, включающей преобразование образа, присущего человеку, в образ, присущий машине, и обратно, может оказаться сравнимой со стоимостью обработки переданной информации. (Поэтому ведутся исследования, цель которых путем автоматического или автоматизированного перевода минимизировать стоимость этой передачи.) Задача обеспечения взаимодействия обычно называется «вводом-выводом».

Человек вводит свою информацию через вводные устройства (например, клавиатуру, дискетки, устройства ввода с перфолент или магнитных лент и другие терминалы), а получает результаты через выводные устройства (печатающие устройства, те же дискетки и магнитные ленты, графопостроители, звуковые и видеодисплеи). Все, что общее для них и что определяется каждой конкретной машиной, — некоторое множество допустимых символов (см. гл. 2). Такое множество символов в Паскале определяется через стандартный тип Text (см. гл. 9).

Необходимо помнить, что каждое из устройств ввода-вывода ориентировано на некоторые соглашения, касающиеся смысла каких-либо символов или их комбинаций. Большинство, например, печатающих устройств ограничивают максимальный размер выводимой строчки. Многие прежние устройства первый символ каждой строчки воспринимают как символ «управления кареткой» и не печатают его. Так можно управлять страницами или требовать надпечатки. Поэтому, представляя каждое устройство через текстовый файл, всегда нужно помнить о соглашениях, обязательных для этого устройства.

С текстовыми файлами можно работать с помощью предописанных файловых процедур Get и Put. Это может показаться обременительным, поскольку процедуры предназначены для манипуляций с одним символом. Как пример рассмотрим ситуацию, когда в переменной X хранится вещественное число и нужно его отпечатать через файл Output. Заметим, что образы символов, обозначающих десятичное представление значения, уже совершенно отличаются от обозначения того же значения римскими цифрами (см. программу 4.9). Однако все, что связано с десятичным представлением, благоразумно передавать встроенным, стандартным процедурам преобразования, переводящим абстрактные числа (в каком бы виде они ни были представлены в машине) в последовательность десятичных цифр и наоборот.

Поэтому две стандартные предописанные процедуры несколько усложнены, чтобы облегчить анализ и формирование текстовых файлов.

12.1. СТАНДАРТНЫЕ ФАЙЛЫ INPUT и OUTPUT

Для представления стандартных устройств ввода-вывода машины (таких, как клавишная панель и видеодисплей) выбраны стандартные текстовые файлы Input и Output. Именно они представляют собой главные линии связи человека с машиной.

Поскольку эти файлы используются очень часто, то именно их и подразумевают в операциях с текстовыми файлами, если, конечно, не указаны какие-либо явные текстовые файлы. Таким образом:

```
Write(Ch)  =  Write(Output,Ch)
Read(Ch)   =  Read(Input,Ch)
Writeln    =  Writeln(Output)
Readln     =  Readln(Input)
Eof        =  Eof(Input)
Eoln       =  Eoln(Input)
Page       =  Page(Output)  (см. раздел 12.4)
```

Если в этих процедурах и функциях параметр-файл не задан, то считается, что к списку параметров как бы *должен* добавляться файл Input или Output.

Внимание: эффект от применения предописанных процедур Reset или Rewrite к файлам Input или Output зависит от реализации.

Таким образом, чтение или запись в любой текстовый файл может проводиться по такой схеме (предполагается, что есть следую-

щие описания: var Ch: Char; B1, B2: Boolean, а P, Q и R — процедуры, определенные пользователем).

Запись символов в файл Output:

```
repeat
   repeat P(Ch); Write(Ch)
   until B1;
   Writeln
until B2
```

Чтение символов из файла Input:

```
while not eof do
   begin { обработка строчки } P;
      while not eoln do
         begin Read(Ch); Q(Ch)
         end;
      R; Readln
   end
```

Ниже приведены два примера программ, показывающие, как пользоваться текстовыми файлами Input и Output. (Разберитесь, какие необходимо провести изменения, если вместо процедур Read и Write будут использоваться только процедуры Get и Put.)

```
program LetterFrequencies(Input,Output);

   { Программа 12.1 — Подсчет частоты встречаемости букв
                      в файле Input.}

   type
      Natural = 0..MaxInt;

   var
      Ch: Char;
      Count: array [Char] of Natural;
      Letters, Upper, Lower: set of Char;

   begin
      Upper := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
                'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
```

```
   Lower := ['a','b','c','d','e','f','g','h','i','j','k','l','m',
             'n','o','p','q','r','s','t','u','v','w','x','y','z'];
   Letters := Lower + Upper;
   for Ch := 'A' to 'Z' do
      Count[Ch] := 0;
   for Ch := 'a' to 'z' do
      Count[Ch] := 0;
   while not Eof do
      begin
         while not Eoln do
            begin  Read(Ch);  Write(Ch);
               if Ch in Letters then Count[Ch] := Count[Ch] + 1
            end;
         Readln;  Writeln
      end;
   for Ch := 'A' to 'Z' do
      if Ch in Upper then Writeln(Ch, Count[Ch]);
   for Ch := 'a' to 'z' do
      if Ch in Lower then Writeln(Ch, Count[Ch]);
end .
```

Дает в качестве результата:

```
A rat in Tom's house might eat Tom's ice cream! (Arithmetic)
Pack my box with five dozen liquor jugs.
The quick brown fox jumped over the lazy sleeping dog.
A        2
B        0
C        0
D        0
E        0
F        0
G        0
H        0
I        0
J        0
K        0
L        0
M        0
N        0
```

```
O       0
P       1
Q       0
R       0
S       0
T       3
U       0
V       0
W       0
X       0
Y       0
Z       0
a       5
b       2
c       5
d       3
e      13
f       2
g       4
h       6
i      10
j       2
k       2
l       3
m       7
n       4
o      10
p       2
q       2
r       6
s       5
t       7
u       5
v       2
w       2
x       2
y       2
z       2
```

Следующая программа копирует Input на Output, добавляя в начало каждой строчки ее номер.

```
program AddIn(Input,Output);

   {Программа 12.2 — В текстовый файл добавляются номера строк.}

   type
      Natural = 0..MaxInt;

   var
      LineNum: Natural;

begin
   LineNum := 0;
   while not Eof do
      begin
         LineNum := LineNum + 1;
         Write(LineNum :4, ' ');
         while not Eoln do
            begin
               Write(Input↑);  Get(Input)
            end;
         Readln;  Writeln
      end
end
```

Дает в качестве результата:

```
1 A rat in Tom's house might eat Tom's ice cream! (Arithmetic)
2 Pack my box with five dozen liquor jugs.
3 The quick brown fox jumped over the lazy sleeping dog.
```

Если файловая переменная Input сопоставлена с входным устройством (таким, например, как клавишная панель), связанным с интерактивным терминалом, то большинство реализаций Паскаля задерживают вычисление буферной переменной Input↑ до тех пор, пока ее значение явно не потребуется в программе. Использование Input↑ в выражении либо неявное ее использование при работе с Read, Readln, eof или eoln вызовет ее вычисление (получение с терминала). Хотя в начале программы и выполняется неявное обращение к Reset(Input), тем не менее программа не будет ожидать прихода данных с терминала до тех пор, пока они

не будут нужны, например, пока не произойдет обращение за Input↑. Если программа дает некоторое сообщение, на которое пользователь должен ответить, и ответ этот надо прочитать, то чтение ответа должно идти уже после того, как будет записано приглашение (все как в обычной жизни).

Приведенный ниже фрагмент программы иллюстрирует процесс выдачи пользователю приглашения через интерактивный терминал:

```
program PromptExample(Input,Output);
   var Guess: Integer;
      .
      .
      .
begin { Здесь идет неявный Reset (Input). }
   Writeln('Please enter an integer between 1 and 10.');
   Read(Guess)
      .
      .
      .
```

В тех реализациях Паскаля, где задержка вычисления не делается, это приведет к ожиданию или требованию данных еще до записи сообщения, поскольку в начале программы было выполнено неявное обращение к Reset(Input). Задерживается вычисление Input↑ или нет — определяется при реализации.

12.2. ПРОЦЕДУРЫ READ и READLN

Процедура Read уже была определена для текстовых файлов в разд. 9.2. В отличие от обычных процедур она воспринимает разное число параметров, и, кроме того, эти параметры могут относиться и к типу Integer (или любому диапазону из Integer), и к типу Real.

Пусть V1, V2, ..., Vn обозначают переменные типа Char, Integer (либо диапазоны из них) или Real, а F — текстовый файл.

Если F не определено, или находится в режиме просмотра, или eof(F) дает значение истина (true), то обращение Read(F, V) — ошибка.

1. Read(V1, ..., Vn) эквивалентно:
   Read(Input, V1, ..., Vn)
2. Read(F, V1, ..., Vn) эквивалентно:
   begin Read(F, V1); ...; Read(F, Vn) end
3. Readln(V1, ..., Vn) эквивалентно:
   Readln(Input, V1, ..., Vn)
4. Readln(F, V1, ..., Vn) эквивалентно:
   begin Read(F, V1); ...; Read(F, Vn); Readln(F) end

Действие Readln заключается в том, что после чтения Vn (из текстового файла F) оставшаяся часть текущей строки пропускается. (Однако сами значения V1, ..., Vn могут располагаться и в нескольких строках.)

5. Если Ch — переменная типа Char или диапазона из Char, то

```
Read(F, Ch) эквивалентно:
begin Ch := F↑; Get(F) end
```

6. Если параметр V относится к типу Integer или же некоторому его диапазону, то Read(F, V) воспринимает последовательность символов, образующую целое число со знаком и, возможно, начальными пробелами. Целое значение, обозначенное такой последовательностью, затем присваивается V.

7. Если параметр V относится к типу Real, то Read(F, V) воспринимает последовательность символов, образующую число со знаком и, возможно, с начальными пробелами. Обозначенное такой последовательностью вещественное значение затем присваивается V.

В процессе просмотра файла F (пропуская пробелы) при чтении числа Read может пропускать и маркеры концов строк. Файл F остается после этого в положении, указывающем на отличный от цифры символ, следующий за последней цифрой, входящей в состав числа. Для правильного чтения последовательных чисел разделяйте их пробелами или располагайте в отдельных строчках. Read воспринимает самую длинную последовательность цифр, и если два числа не отделить друг от друга, то Read не сможет разделить их (и человек не сможет!).

Примеры.

Считывание и обработка последовательности чисел, за последним из которых идет символ «звездочка». Считается, что F — текстовый файл, а X и Ch — переменные, относящиеся соответственно к типу Integer (или Real) и Char.

```
Reset(F);
repeat
   Read(F,X,Ch);
   P(X)
until Ch='*'
```

Одна из наиболее часто встречающихся ситуаций заключается в том, что мы не знаем, сколько чисел нужно прочитать, а специального символа, маркирующего конец последовательности, нет. Ниже приводятся две подходящие для этого случая схемы. В них используется процедура SkipBlanks:

```
procedure SkipBlanks(var F: Text);
   var Done: Boolean;
begin
   Done := False;
   repeat
      if eof(F) then Done := True
      else
         if F↑ = ' ' then Get(F)
         else Done := True
   until Done
end
```

В первой схеме числа обрабатываются поодиночке:

```
Reset(F);
while not eof(F) do
   begin
      Read(F,X); SkipBlanks(F);
      P(X);
   end
```

Во второй схеме обрабатывается по n чисел:

```
Reset(F);
while not eof(F) do
   begin
      Read(F,X1,...,Xn); SkipBlanks(F)
      P(X1,...,Xn);
   end
```

(Для правильной работы этой схемы необходимо, чтобы количество чисел было кратным n.)

12.3. ПРОЦЕДУРЫ WRITE и WRITELN

Процедура Write для текстовых файлов уже была определена в разд. 9.2. В отличие от обычных процедур она воспринимает разное число параметров, и, кроме того, эти параметры могут относиться к типам, совместимым с типами Integer, Real, Boolean, или строковым типам.

Процедура Write добавляет к текстовым файлам строки символов (состоящие из одного или более символов). Пусть P1, P2, ..., Pn — параметры, вид которых определен синтаксической диаграм-

мой на рис. 12.1, а F — текстовый файл. Если F не определено, или этот файл не находится в режиме формирования, или же eof(F) дает значение, не равное true, то обращение Write(F, P) — ошибка.

1. Write(P1, ..., Pn) эквивалентно:
   Write(Output, P1, ..., Pn)
2. Write(F, P1, ..., Pn) эквивалентно:
   begin Write(F, P1); ...; Write(F, Pn) end
3. Writeln(P1, ..., Pn) эквивалентно:
   Writeln(Output, P1, ..., Pn)
4. Writeln(F, P1, ..., Pn) эквивалентно:
   begin Write(F, P1); ...; Write(F, Pn); Writeln(F) end

Причем Writeln сначала записывает P1, ..., Pn, а затем заканчивает текущую строчку текстового файла F.

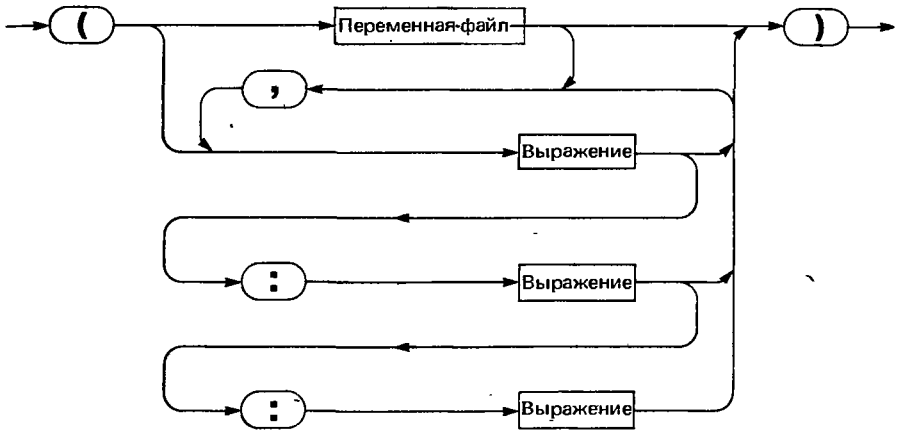


Рис. 12.1. Синтаксическая диаграмма
для *Списка параметров вывода*

5. Каждый из параметров Pi должен выглядеть так:

e
e: w
e: w: f

где e, w и f — выражения, причем e — это то значение, которое нужно записать (его тип — Char, Integer, Boolean, Real или любая

строчка), w называется минимальным *размером поля* и включается в обращение по желанию. Оно должно быть положительным целым значением; указывает число записываемых символов. Как правило, e записывается с помощью w символов (перед ним, если надо, ставятся пробелы). Если размер поля не указан, то ему, в соответствии с типом e, приписывается некоторое значение. Выражение f называется размером *дробной части* и используется по желанию в том случае, если e типа Real. Это должно быть положительное целое выражение.

6. Если e типа Char, то подразумеваемое значение w — 1. Поэтому Write(F, C) эквивалентно f1 := C; Put(F).

7. Если e типа Integer, то подразумеваемое значение w определяется при реализации. Если даже значение w мало и его не хватает для размещения целого, тем не менее все представление целого будет записано (включая, если необходимо, и «—»).

8. Если e относится к строковому типу, то подразумеваемое значение w равно длине строки. Если же w меньше размера строки, то будет записано ровно w символов из строки e.

9. Если тип e — Boolean, то подразумеваемое значение w определяется при реализации. В зависимости от значения e по правилам п. 8 будет записана одна из строк: 'true' или 'false'. Будут ли при этом для представления данных строк использованы прописные либо строчные буквы (или даже их комбинации) — определяется при реализации.

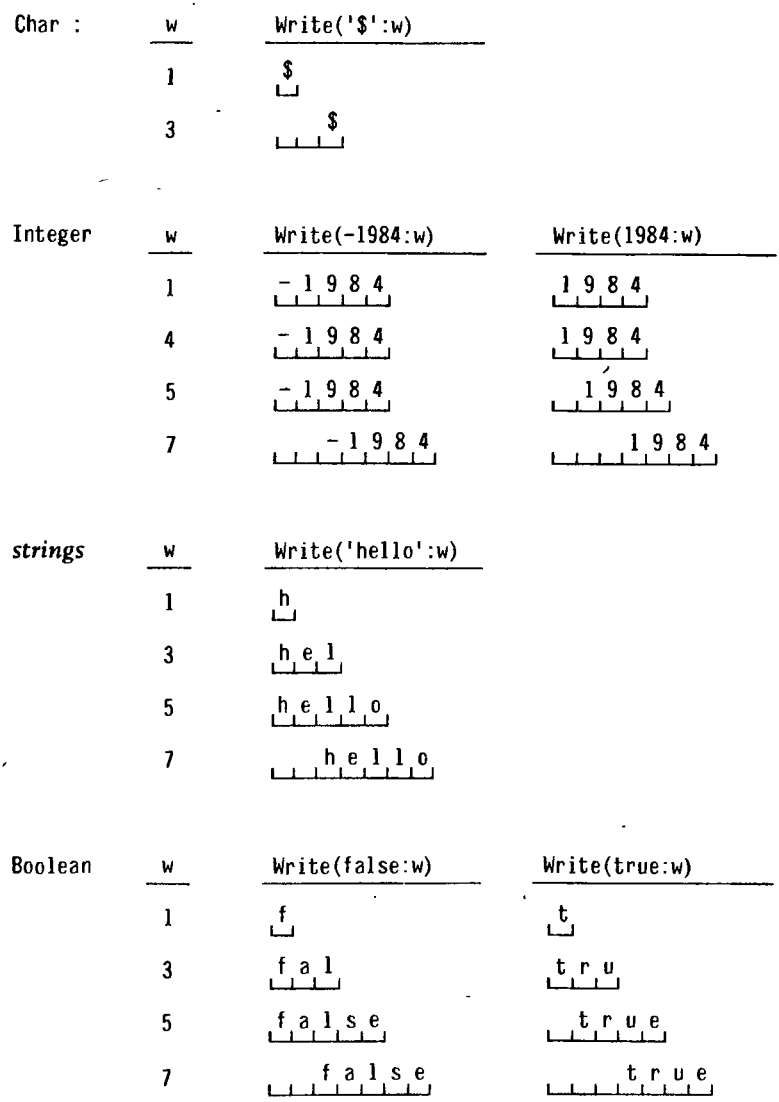
10. Если тип e — Real, то подразумеваемое значение w определяется при реализации. Если w меньше числа символов, требующихся для записи вещественного числа, то выделяется столько места, сколько нужно (выделяется даже пространство для символа «—», если e — отрицательное). Если задано f (размер дробной части), то значение e записывается в форме с фиксированной запятой. Если же f не указано, то число записывается в десятичном виде с плавающей запятой и порядком.

В общем виде запись с фиксированной запятой представляет собой такую последовательность символов: возможный знак минус (если число отрицательное), последовательность цифр, представляющая целую часть, точка (десятичная запятая) и последовательность, представляющая собой дробную часть. Размер дробной части определяется f.

Запись с плавающей запятой в общем виде выглядит так: пробел или знак минус, одна цифра, точка (десятичная запятая), последовательность цифр, буква e (или E), знак плюс или минус и последовательность цифр, представляющая порядок, размер которой определяется при реализации. Длина первой последователь-

ности цифр (предшествующей букве e) варьируется в зависимости от значения w.

На рис. 12.2 приводятся примеры вывода каждого из типов.



Real	w	f	Write(123.789:w:f)	Write(-123.789:w:f)
	1	1	123.8	-123.8
	1	3	123.789	-123.789
	1	4	123.7890	-123.7890
	5	1	123.8	-123.8
	6	1	123.8	-123.8
	7	1	123.8	-123.8
		w	Write(987.6:w)	Write(-987.6:w)
		1	9.9E+02	-9.9E+02
		8	9.9E+02	-9.9E+02
		9	9.88E+02	-9.88E+02
		10	9.876E+02	-9.876E+02
		11	9.8760E+02	-9.8760E+02

Рис. 12.2. Примеры форматного вывода

12.4. ПРОЦЕДУРА PAGE

Для удобства работы с текстовыми файлами в Паскале существует предописанная процедура Page. Обращение к ней приводит к тому, что при последующей выдаче (при печати F или выводе на дисплей), текст, записанный после обращения, появится на новой «странице». Действие Page(F) на файл F определяется при реализации. Чаще всего при обращении Page(F) в файл записывается некоторый управляющий символ (например, в множестве символов ASCII это ff — Form Feed), который и приводит к желаемому эффекту.

Замечание. Если перед обращением к Page(F) не было выполнено Writeln, то в качестве своей первой работы Page(F) и выполняет такое неявное обращение. Файл F должен быть определен и находиться в режиме формирования, если это не так, то обращение Page(F) — ошибка. Что происходит с файлом при чтении, если к нему применялись обращения Page(F) — зависит от реализации.

ОПИСАНИЕ ЯЗЫКА

1. ВВЕДЕНИЕ

Создание языка Паскаль преследовало две основные цели. Во-первых, хотелось построить язык, пригодный для обучения программированию как некоторой систематической дисциплине, основанной на нескольких фундаментальных понятиях, причем эти понятия должны были найти естественное и ясное отражение в языке. Во-вторых, хотелось реализовать этот язык на современных машинах надежным и эффективным способом.

Желание обратиться при обучении программированию к новому языку объясняется нашей неудовлетворенностью большинством используемых сейчас языков. Особенности и конструкции этих языков часто невозможно логически и убедительно объяснить; как правило, они при систематическом подходе вызывают нарекания. Кроме того, мы убеждены, что язык, на котором студент учится выражать свои мысли, оказывает глубокое влияние на его изобретательность и способ мышления, поэтому царящий в существующих языках беспорядок непосредственно сказывается на стиле программирования студентов.

Конечно, вводить еще один язык программирования следует весьма осмотрительно, и, кроме того, обучение программированию на языке, не распространенном широко и не всем доступном, вызывает возражения, обусловленные до некоторой степени сиюминутными коммерческими соображениями. Однако, если выбирать язык для обучения, исходя только из его распространенности и доступности, мы обречем себя на застой, поскольку язык, которому мы все больше будем обучать, станет из-за этого еще более распространенным. Такие соображения показались нам вполне заслуживающими того, чтобы попытаться разорвать этот порочный круг.

Новый язык, безусловно, не следует создавать только ради новизны. В качестве его основы нужно использовать уже существующие языки, если они удовлетворяют упомянутым критериям и не препятствуют систематическому подходу. В этом смысле в качестве основы Паскаля был использован Алгол-60, так как он больше подходит для обучения, чем другие стандартные языки. Из Алго-

ла-60 были «скопированы» принципы создания сложных структур и фактически вид выражения. Однако мы не сочли возможным и целесообразным «встраивать» Алгол-60 как подмножество Паскаля: некоторые принципы его построения, особенно связанные с описаниями, оказались бы несовместимы с принципами, обеспечивающими естественное и удобное представление новых конструкций в Паскале.

Основные расширения языка Паскаль по сравнению с Алголом-60 связаны с возможностями построения данных сложной структуры; отсутствие таких возможностей в Алголе-60 считается главной причиной относительной узости области его применения. Введение записей и файлов позволило решать с помощью Паскаля задачи коммерческого типа или по крайней мере демонстрировать такие задачи в курсе программирования.

2. ОБЗОР ЯЗЫКА

Любая программа для вычислительной машины состоит из двух основных частей: описания *действий,* которые следует выполнить, и описания *данных*, с которыми манипулируют эти действия. Действия задаются так называемыми *операторами,* а данные — *описаниями* и *определениями.*

Данные представляются с помощью значений *переменных.* Каждая встречающаяся в некотором операторе переменная должна вводиться через *описание переменной,* связывающее с этой переменной имя и тип данных. *Тип* фактически определяет множество значений, которые может принимать переменная, и ограничивает множество операций, которые можно над нею выполнять. В Паскале тип можно либо явно определить в описании переменной, либо с помощью *описания типа* сопоставить с ним некоторое имя типа и затем уже ссылаться на этот тип через это имя.

Простые типы состоят из предопределенного типа Real (вещественный) и различных предопределенных ординальных* типов. Каждый простой тип определяет некоторое упорядоченное множество значений. Любой же ординальный тип характеризуется взаимно однозначным отображением его значений на некоторый интервал целых чисел — так называемые ординальные номера этих значений.

Основными ординальными типами являются определяемые программистом перечисляемые (enumerated) типы и предопреде-

* См. сноску на с. 24. — *Примеч. пер.*

ленные типы — Boolean (логический), Char (символьный) и Integer (целый). Перечисляемый тип вводит новое множество значений, причем каждое из значений обозначается некоторым именем, отличным от других имен. Значения типа Char обозначаются с помощью «закавычивания» символов, а вещественные и целые значения — с помощью чисел (последние синтаксически отличаются от имен). Множества значений типа Char и их графическое представление варьируются от реализации к реализации и зависят от множества символов, принятого на каждой конкретной машине.

Ординальные типы можно определять и как *диапазоны* любых основных ординальных типов (базовых типов). Для этого нужно указать самое маленькое и самое большое значения из интервала значений, относящихся к этому диапазону.

Составные (structured) типы определяются путем описания типов их компонент и указания метода объединения (компонент в единое целое. — *Примеч. пер.*). Методы объединения отличаются один от другого механизмами, позволяющими обращаться к компонентам переменной составного типа. В Паскале существуют четыре основных метода объединения, порождающие соответственно такие структуры данных: массивы, записи, множества и файлы.

В *массивах* все компоненты одного типа. Обращение к ним происходит с помощью вычисляемых *индексов*, тип которых указывается в описании самого массивового типа. Индексы должны относиться к ординальному типу. Обычно это некоторый перечисляемый тип или диапазон из целого типа. Если задать значение, относящееся к типу индекса, то переменная с индексом укажет одну из компонент массива. Поэтому всякую переменную-массив можно рассматривать как некоторое отображение типа индекса в тип компонент. Время, затрачиваемое на обращение к одной компоненте, не зависит от значения индекса. Отсюда и структуры класса «массив» называются структурами с *произвольным доступом*.

В *записях* компоненты (называемые *полями*) не обязательно относятся к одному типу. Для того чтобы тип поля был очевиден из текста программы (не прибегая к выполнению программы), поля определяются не через вычисляемые значения, а просто с помощью уникальных имен. Такие имена полей указываются в описании типа. Время, необходимое для обращения к какой-либо компоненте, опять же не зависит от имени поля, и записи поэтому относятся к структурам с произвольным доступом.

Можно задавать записной тип с несколькими *вариантами*. Это означает, что различные переменные, хотя про них и говорят, что они относятся к одному типу, могут иметь несколько отличную друг от друга структуру. Разница может заключаться и в числе, и в типе

компонент. Вариант, к которому относится текущее значение записи, может указываться посредством поля, общего для всех вариантов и называемого *полем признака.* Обычно общая для всех вариантов часть состоит из нескольких компонент, в которые включается и поле признака.

Всякий множественный тип определяет множество значений, представляющее собою множество-степень его базового типа (основания). Базовым типом должен быть ординальный тип и обычно это бывает какой-нибудь перечисляемый тип, тип Char или диапазон из целых чисел. Компоненты (элементы) множества прямо не доступны, однако предусмотренные операции над множествами (включающие проверку принадлежности значения множеству) и конструкторы множеств позволяют и порождать множества, и манипулировать ими.

Структуры класса *«файл»* представляют собою *последовательность* компонент одного типа. В последовательности определен некоторый естественный порядок компонент. В любой момент непосредственно доступна только одна компонента. Ее можно либо «просматривать», либо «формировать»; одновременно делать то и другое невозможно. К остальным компонентам можно «прийти», лишь двигаясь шаг за шагом вдоль файла. Файл формируется посредством последовательных добавлений в его конец новых компонент. Поэтому определение файлового типа не задает число компонент.

Описание переменной связывает с именем некоторый тип, и в момент активации блока (см. ниже), где встретилось такое описание, порождается переменная, идентифицируемая упомянутым именем. Переменные, описываемые такими явными описаниями, иногда называют *статическими.* Кроме того, переменную можно создать, выполняя некоторый оператор; такое *динамическое* формирование дает то, что называется *ссылкой* (заменяющей явное имя). Впоследствии эта ссылка используется для идентификации созданной переменной. Значение ссылки можно присваивать переменной или функции, относящимся к типу этой ссылки. Любой ссылочный тип имеет фиксированный тип *области,* и каждая переменная, идентифицированная некоторым ссылочным значением, относящимся к определенному ссылочному типу, принадлежит его типу области. Кроме таких *идентифицирующих* значений, в любой ссылочный тип включается и значение nil, не указывающее ни на какую переменную. Поскольку компоненты составных переменных могут относиться к ссылочным типам, а типы области этих ссылочных типов в свою очередь могут быть составными, то с помощью ссылок можно представлять конечные графы во всей их полноте.

Среди операторов наиболее важный — оператор *присваивания*. Он указывает, что значение, полученное при вычислении некоторого *выражения*, необходимо присвоить некоторой переменной (или ее компоненте). Выражение состоит из переменных, констант, границ индексов параметров-массивов, конструкторов множеств и операций, а также функций, применяемых к указанным величинам и дающих новые, результирующие значения. Переменные, константы и функции либо описываются в программе, либо относятся к стандартным («предописанным») объектам. В Паскале определяется фиксированное множество операций, каждую из которых можно рассматривать как отображение типов операндов в тип результата. Множество операций делится на следующие группы:

1. *Арифметические* операции — сложение, вычитание, изменение знака, умножение, деление и вычисление остатка.

2. *Логические* операции — отрицание, объединение (or) и конъюнкция (and).

3. Операции *над множествами* — объединение, пересечение, разность.

4. Операции *отношения* — равенство, неравенство, упорядоченность, принадлежность к множеству и включение. Тип результата операции отношения — Boolean.

Оператор *процедуры* вызывает выполнение указанной процедуры (см. ниже). Операторы присваивания и процедуры фигурируют в качестве компонент или «строительных блоков» *сложных* операторов, которые задают последовательное, выборочное либо повторяющееся выполнение своих составляющих. Последовательное выполнение задается *составным* оператором, условное или выборочное — *условным* оператором, или оператором *варианта*, повторяющееся — оператором цикла с *предусловием*, *с постусловием* и *с параметром* (шагом). Условный оператор выполняет (или не выполняет) оператор в зависимости от значения логического выражения, а оператор варианта позволяет в соответствии со значением ординального выражения выбрать из многих операторов один. Цикл с шагом используется для выполнения оператора-компоненты, причем каждый раз управляемой переменной (параметру цикла) присваивается очередное ординальное значение. В других же случаях используются циклы с предусловиями и постусловиями.

Кроме того, в Паскале предусмотрены и операторы *перехода*, указывающие, что выполнение должно продолжаться с другого места программы; это место маркируется *меткой*. Метка должна быть описана.

Операторы и описания меток, констант, типов, переменных, процедур и функций объединяются вместе в *блоки*. На метки, кон-

станты, переменные, процедуры и функции, описанные в некотором блоке, можно ссылаться только внутри данного блока, поэтому они называются *локальными* по отношению к данному блоку. Их имена имеют смысл только в тексте программы, составляющем соответствующий блок, и называемом *областью действия* (scope) этих имен. Блок же лежит и в основе описаний *программ, процедур* и *функций*; в этих случаях блоку дается некоторое имя, с помощью которого можно обозначать такой блок. Так как процедуры и функции могут вкладываться одна в другую, то вложенными могут быть и области действия.

Процедура или функция имеет фиксированное число параметров, каждый из которых обозначается внутри процедуры или функции с помощью имени, называемого *формальным* параметром. При активации процедуры или функции для каждого из параметров должна быть указана фактическая величина, на которую можно ссылаться изнутри процедуры или функции через соответствующий формальный параметр. Эта величина называется *фактическим* параметром. Параметры бывают четырех видов: параметры-значения, параметры-переменные, параметры-процедуры и параметры-функции. В первом случае фактический параметр — это выражение, которое вычисляется, а его значение присваивается формальному параметру; все это проделывается один раз перед началом каждой активации процедуры или функции. Формальный параметр в этом случае представляет собой локальную переменную. В случае параметра-переменной фактический параметр обозначает некоторую переменную и формальный параметр во время всей активации процедуры или функции обозначает ту же переменную. В случае параметра-процедуры (процедурального) или параметра-функции (функционального) фактический параметр — имя процедуры или функции.

Функции описываются аналогично процедурам, но функции дают еще результат, относящийся к типу, который должен быть задан в описании функции. По соглашению тип результата должен быть простым или ссылочным. Функции можно употреблять как составные части в выражениях. Внутри описания функции необходимо избегать присваиваний нелокализованным переменным и других так называемых побочных эффектов.

3. НОТАЦИЯ И ТЕРМИНОЛОГИЯ

Синтаксические конструкции обозначаются английскими словами (метаименами), напечатанными курсивом, и определяются в соответствии с Расширенными Бэкуса—Наура Формами

(РБНФ) [13]. Каждое правило РБНФ определяет метаимя с помощью РБНФ-выражения, состоящего из одной или более альтернатив (фраз), разделенных вертикальной чертой (|). Фраза состоит из нуля или более элементов. Элемент — это или некоторое метаимя, или некоторая буквальная комбинация символов, заключенная в кавычки («»), или другое РБНФ-выражение, заключенное с двух сторон в фигурные, квадратные или круглые скобки. Фигурные скобки { } указывают на повторение (нуль или более вхождений), квадратные скобки [ ] — на допустимость (нуль или одно вхождение), а круглые скобки ( ) указывают на группирование (точно одно вхождение) выражений, в них заключенных.

В разд. 4 правила РБНФ описывают формирование из отдельных символов целых *лексем;* в лексему не должны входить дополнительные символы. В разд. 5—13 правила РБНФ определяют синтаксис программ в терминах лексем; лексемы могут разделяться (одна от другой) символами-разделителями (как это описано в разд. 4).

Термин «ошибка» (error) относится к действию или состоянию программы, нарушающим стандарт, и к таким, что любой конкретный процессор не всегда их может обнаружить.

Выражение *«определяется при реализации»* означает, что некоторые конструкции языка Паскаль в различных реализациях могут быть отличными одна от другой, причем в каждой из реализаций должно быть определено, как эта конструкция реализована.

Выражение *«зависит от реализации»* означает, что некоторая конструкция в разных реализациях сделана по-разному, но при реализации не определяется, как это сделано.

Расширение — это дополнительная конструкция или свойство, недоступная во всех реализациях, причем она не действует на конструкции стандарта Паскаля. Реализации часто поддерживают расширения в форме дополнительных предопределенных и предописанных констант, типов, переменных, процедур и функций.

Любая программа, соответствующая стандарту, не должна включать зависящие от реализации конструкции или какие-либо расширения. В любой же переносимой программе следует, кроме всего прочего, внимательно следить за употреблением конструкций, определяемых при реализации (например, множеством символов или диапазоном изменения целых значений).

4. ЛЕКСЕМЫ И СИМВОЛЫ-РАЗДЕЛИТЕЛИ

Любая программа выглядит как последовательность лексем, расположенных в соответствии с правилами и синтаксисом Паска-

ля. Соседние лексемы часто отделяются одна от другой для удобочитаемости символами-разделителями. Все лексемы делятся на специальные лексемы, имена, директивы, числа, метки и строки символов. Символы-разделители — это пробелы, примечания и концы строк (в текстовом представлении программ).

```
Специальные-символы = "+" | "-" | "*" | "/" |
                      "=" | "<>" | "<" | "<=" | ">" | ">=" |
                      "(" | ")" | "[" | "]" | ":=" | "." | ".." |
                      ":" | "," | ";" | "↑" | Символы-слова.

Символы-слова = "div" | "mod" | "nil" | "in" | "or" | "and" |
                "not" | "if" | "then" | "else" | "case" | "of" |
                "repeat" | "until" | "while" | "do" | "for" |
                "to" | "goto" | "downto" | "begin" | "end" |
                "with" | "const" | "var" | "type" | "array" |
                "record" | "set" | "file" | "function" |
                "procedure" | "label" | "packed" | "program".
```

В стандарте предусмотрены и такие альтернативные представления:

Основной символ	Альтернативное представление
↑	^ или @
[	(.
]	.)

Многие из лексем строятся из букв и цифр. Везде, за исключением «внутренности» строки символов, строчные буквы эквивалентны соответствующим прописным.

```
Буква = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
        "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
        "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
        "y" | "z"

Цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Имена предназначены для обозначения констант, типов, переменных, процедур, функций и границ. Директивы же употребляются в описаниях процедур и функций.

Имя = *Буква* { *Буква* | *Цифра* }.
Директива = *Буква* { *Буква* | *Цифра* }.

Написание символов-слов, имен или директив — это вся последовательность, состоящая из входящих в нее букв и цифр. Ни одно имя или директива не могут иметь написание, совпадающее с написанием символов-слов.

Примеры имен (шесть различных написаний):

```
FirstPlace    ord    ProcedureOrFunctionDeclaration
Elizabeth     John   ProcedureOrFunctionHeading
```

С помощью описания или определения вводится написание конкретного имени, и ему приписывается определенный смысл. Смысл, связанный с этим фиксированным написанием, остается неизменным внутри некоторой части текста программы, называемой *областью действия* (scope) данного описания или определения (см. разд. 10).

Для чисел используется традиционная десятичная система записи. Целые и вещественные числа без знака — константы, относящиеся соответственно к предопределенным типам Integer и Real (см. разд. 6.1.2). Буква «e», предшествующая в вещественных числах без знака порядку, означает «умножить на 10 в степени». Максимальное значение целого числа без знака может быть задано с помощью значения предопределенной константы Maxint, определенной при реализации.

Число без знака = *Целое без знака* | *Вещественное без знака*.
Целое без знака = *Последовательность цифр*.
Вещественное без знака = *Целое без знака* "." *Последовательность цифр* ["e" *Порядок*] |
Последовательность цифр "e" *Порядок*.
Порядок = [*Знак*] *Целое без знака*.
Знак = "+" | "−".
Последовательность цифр = *Цифра* {*Цифра*}.

Примеры целых без знака:

```
1        100        00100
```

Примеры вещественных без знака:

```
0.1      0.1e0      87.35e + 8      1E2
```

Числовой ввод из текстовых файлов может воспринимать и числа со знаком (см. разд. 12).

Число со знаком = *Целое со знаком* | *Вещественное со знаком*.
Целое со знаком = [*Знак*] *Целое без знака*.
Вещественное со знаком = [*Знак*] *Вещественное без знака*.

Строка символов — это заключенная в апострофы последовательность элементов строки. Любой элемент строки представляет определяемое при реализации значение предопределенного типа Char; элемент состоит либо из двух идущих подряд апострофов, либо из любого другого, определяемого реализацией символа. Два разных символа, фигурирующие в качестве элементов строки, должны обозначать различные значения типа Char. Элемент строки, состоящий из двух апострофов, обозначает символ апострофа.

Строка символов = «'» *Элемент строки* { *Элемент строки* } «'».
Элемент строки = «''» | *Любой символ кроме апострофа.*

Строка символов, если она содержит один элемент строки, относится к константам типа Char, в противном же случае — к константам строкового типа (см. разд. 6.2.1), имеющим столько компонент, сколько было в строке элементов.

Замечание. Любая строка символов должна записываться точно в одной строке текста программы.

Примеры строк символов:

```
 A'          ':'
'Pascal'     ''''
'This is a character string'
```

Между любыми двумя соседними лексемами и перед первой лексемой программы могут помещаться символы-разделители. Между двумя соседними именами, директивами, служебными словами (символами-словами), метками или числами должен находиться по крайней мере один символ-разделитель. Разделитель — это пробел, конец строки текста программы или примечание. Смысл любой программы от замены любого примечания на пробел не изменяется.

Примечание = ("{" | "(*") {*Элемент примечания*} ("}" | "*)").

Элемент примечания — это либо конец строки, либо любая последовательность символов, не содержащая "}" или "*)".

Замечание. Допускаются примечания {... *} и (* ... }. Примечание { (*) эквивалентно примечанию {(}.

5. КОНСТАНТЫ

Определение константы вводит имя константы как обозначение для величины константы из определения; определяемое имя константы не должно входить в константную часть данного опреде-

ления*. Определения констант объединяются в раздел определения констант.

Раздел определения констант = ["const" *Определение константы* ";" {*Определение константы* ";"}].
Определение константы = *Имя* "=" *Константа*.
Константа = [*Знак*] (*Число без знака*|*Имя константы*)|
Строка символов.
Имя константы = *Имя*.

Имя константы, перед которым стоит знак («+» или «—»), должно обозначать значение типа Integer или Real.

Существует три стандартных, предопределенных имени констант: Maxint обозначает определяемое реализацией значение типа Integer, а false и true — значения логического типа (см. разд. 6.1.2.).

Пример раздела определения констант:

```
const
   N = 20;
   SpeedOfLight = 2.998e8 { meters / second };
   PoleStar = 'Polaris';
   eps = 1E-6;
```

6. ТИПЫ

Любой тип определяет множество значений переменных, выражений, функции и т. п., которые относятся к этому типу. Правила совместимости (compatibility) типов определяют совместное использование типов в выражениях, присваиваниях и т. п.

Определение типа вводит для обозначения некоторого типа имя типа. Определяемое имя не должно встречаться в типовой части данного определения; исключение делается лишь для типа области в случае ссылочного типа (см. разд. 6.3). Определения типов объединяются в раздел определения типов. Пример раздела определения типов приводится в разд. 6.4.

Раздел определения типов = ["type" *Определение типа* ";" {*Определение типа* ";"}].
Определение типа = *Имя* "=" *Тип*.
Имя типа = *Имя*.

Типы представляются с помощью РБНФ с метаименем *Тип*. Если представление некоторого типа состоит из одного имени типа,

* Не должно быть рекурсивных определений. — *Примеч. пер.*

то оно представляет некоторый (существующий) тип, который обозначается упомянутым именем типа. Если представление типа состоит не только из имени типа, то оно представляет полностью новый тип. Типы классифицируются в соответствии с некоторыми их свойствами.

Тип = Простой тип | Составной тип | Ссылочный тип.

6.1. ПРОСТЫЕ ТИПЫ

Любой простой тип определяет некоторое упорядоченное множество значений и представляет собою либо предопределенный тип Real *(вещественный)*, либо некоторый *ординальный* (ordinal) тип. Имя вещественного типа — это имя типа, обозначающего тип

Простой тип = Ординальный тип | Имя вещественного типа.
Имя вещественного типа = Имя типа.

Ординальный тип отличается (от типа Real) тем, что существует соответствие один к одному между его значениями и множеством *порядковых (ординальных) чисел.* Порядковые числа для любого ординального типа относятся к некоторому интервалу целых чисел.

К любому ординальному значению X применимы три следующие предописанные функции:

ord(X)	дает ординальное число, соответствующее X; результат относится к типу Integer
succ(X)	дает следующее за X значение, т. е. succ(X) > X и ord(succ(X)) = ord(X) + 1, если только X не максимальное число соответствующего типа. В последнем случае succ(X) суть ошибка
pred(X)	дает значение, предшествующее X, т. е. pred(X) < X и ord(pred(X)) = ord(X) — 1, если только X не минимальное число соответствующего типа. В последнем случае pred(X) суть ошибка

Ясно, что порядок значений любого ординального типа тот же, что и порядок соответствующих им ординальных чисел.

Ординальный тип — это либо *перечисляемый* тип (enumerated), либо один из предопределенных типов Integer, Char или Boolean, либо *диапазон* одного из этих типов.

Ординальный тип = Перечисляемый тип | Диапазонный тип | Имя ординального типа.
Имя ординального типа = Имя типа.

Имя ординального типа — это имя типа, обозначающего ординальный тип*.

6.1.1. Перечисляемые типы. Перечисляемый тип определяет множество полностью новых значений и вводит имена констант, обозначающих каждое из таких значений.

Перечисляемый тип = "("*Список имен*")".
Список имен = *Имя* {"," *Имя*}.

Первое из имен обозначает самое маленькое из значений, ему соответствует ординальное число нуль. Любое другое имя из списка обозначает значение, следующее за значением, обозначенным предшествующим именем. Таким образом, имена констант перечисляются в порядке увеличения значений.

Примеры перечисляемых типов:

```
(Red, Orange, Yellow, Green, Blue)
(Club, Diamond, Heart, Spade)
(Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
```

6.1.2. Предопределенные простые типы. В Паскале стандартными считаются такие имена предопределенных типов:

Real	определяет зависящее от реализации подмножество вещественных чисел
Integer	включает множество целых с абсолютными значениями, меньшими или равными определенного при реализации значения с предопределенным именем константы Maxint**
Boolean	Для любого целого I, ord(I) = I определяет множество истинностных значений, обозначаемых предопределенными именами констант false и true. Заметим, что: false < true и ord(false) = 0

* Эта и аналогичные фразы порождены, похоже, тем, что в английском языке не всегда можно точно фиксировать смысл некоторых конструкций. В частности, выражение «ordinal type identifier» можно понимать и как «ординальное имя типа», и как «имя ординального типа». Конечно, чаще всего «ординальных имен» не бывает, но ведь речь идет о формальном определении языка, и читатель уже привык к тому, что в таких определениях имя «черное» может обозначать значение «белое». Система управления, принятая в русском языке, делает такие фразы тавтологиями. — *Примеч. пер.*

** Если вспомнить предыдущее примечание, то фразу «predefined constant identifier» можно переводить и как «предопределенное имя константы», и как«имя предопределенной константы». А что действительно предопределено? Имя или константа? — *Примеч. пер.*

Char — задает определяемое при реализации множество символов с определяемыми при реализации ординальными числами, такими, что:

а) цифры '0', '1', ..., '9' упорядочены как числа и идут одна за другой (т. е. succ('0') = '1');

б) если есть строчные буквы ('a', 'b', ..., 'z'), то они упорядочены в алфавитном порядке (но не обязательно следуют точно одна за другой);

в) если есть прописные буквы ('A', 'B', ..., 'Z'), то они упорядочены в алфавитном порядке (и не обязательно точно следуют одна за другой).

6.1.3. Диапазонные типы. Множество значений, определяемых диапазонным типом, представляет собою подмножество значений другого ординального типа, он называется базовым типом данного диапазонного типа. Диапазонный тип задается самым маленьким и самым большим значениями и включает все значения, лежащие между ними.

Диапазонный тип = Константа ".." Константа.

Обе константы должны относиться к базовому типу. Первая константа задает самое маленькое значение; она должна быть меньше или равна второй константе, задающей самое большое значение.

Примеры диапазонных типов:

```
1..N
-10 .. +10
Monday..Friday
```

6.2. СОСТАВНЫЕ ТИПЫ

Любой составной тип характеризуется типами его компонент и методом их объединения. Кроме того, для каждого составного типа может быть указано предпочтительное представление данных. Если перед составным типом поставлен префикс packed, то на смысл программы это не оказывает никакого влияния, а лишь подсказывает транслятору, что надо экономить память, выделяемую для значений этого типа, даже за счет эффективности доступа к ним и возможного увеличения объема памяти самой программы. Есть только два исключения: всегда упаковываются строковые типы (см. разд. 6.2.1), а фактические параметры-переменные (см. разд. 11.3) не должны быть компонентами какой-либо упакованной составной переменной. Если компоненты упакованного составного типа относятся к некоторому составному типу, то тип компонент считается упакованным только в том случае, если в представлении типа компонент стоит явный префикс

Составной тип = ["packed"] *Неупакованный составной тип*|
Имя составного типа.
Неупакованный составной тип = *Массивовый тип*|*Записной тип*|
Файловый тип|*Множественный тип.*
Имя составного типа = *Имя типа.*

Имя составного типа есть имя типа, обозначающего составной тип.

6.2.1. Массивовый тип. Массивовый тип относится к структурам, состоящим из фиксированного числа компонент, причем все компоненты принадлежат к одному и тому же типу, называемому типом компонент. Компоненты находятся во взаимо однозначном соответствии со значениями, относящимися к типу индекса.

Массивовый тип = "array" "[" *Тип индекса* {"," *Тип индекса*} "]"
"of" *Тип компоненты.*
Тип индекса = *Ординальный тип.*
Тип компоненты = *Тип.*

Можно задавать более одного типа индекса, как, например, в
packed array [T1, T2, ..., Tn] of C
Это просто сокращение записи:
packed array [T1] of array [T2, ..., Tn] of C.

Эти две записи должны считаться эквивалентными и при отсутствии в них префикса packed.

Примеры массивовых типов:

```
array [1..100] of Real
array [1..10, 1..20] of 0.99
array [Boolean] of Color
array [Size] of packed array ['a'..'z'] of Boolean
```

Каждое значение массивового типа — это функциональное (много к одному) отображение всего множества значений индексов во множество значений, относящихся к типу компонент.

Массивовый тип называется *строковым типом*, если он упакованный, его компоненты относятся к предопределенному типу Char, а тип его индекса — диапазон из Integer от 1 до n, причем n больше 1. Строки символов (см. разд. 4) относятся к константам строковых типов.

Примеры:

```
packed array [1..StringLength] of Char
packed array [1..2] of Char
```

6.2.2. Записные типы. Записной тип имеет фиксированное число компонент, относящихся, возможно, к различным типам. Отдельные компоненты и их типы, а также сами значения записного типа определяются с помощью *списка полей* (field list) из записного типа.

Записной тип = "record" *Список полей* "end"
Список полей = [(*Фиксированная часть* [";" *Вариантная часть*] |
Вариантная часть) [";"]].
Фиксированная часть = *Секция записи* {";" *Секция записи*}.
Секция записи = *Список имен* ":" *Тип*.
Имя поля = *Имя*.

Список полей может содержать *фиксированную часть* (fixed part), которая задает фиксированное число компонент, называемых *полями* (fields). С каждой секцией записи связан некоторый тип: считается, что все имена из ее списка имен полей принадлежат к этому типу. Область действия имен полей включает весь записной тип, а также обозначения полей и операторы присоединения, где эти имена могут использоваться (см. разд. 7.2.2; 9.2.4 и 10.2). Таким образом, в любом из записных типов написание каждого имени поля должно быть уникальным.

Примеры записных типов с одной фиксированной частью.

```
packed record
   Year: 1900..2100;
   Month: 1..12;
   Day: 1..31
end
```

```
record
   Firstname, Lastname:
      packed array [1..32] of Char;
   Age: 0..99;
   Married: Boolean
end
```

Список полей может также содержать и *вариантную часть*, задающую один и более вариантов. Структура и значения варианта задаются его списком полей.

Вариантная часть = "case" *Селектор варианта* "of" *Вариант* {";" *Вариант*}.
Вариант = *Константа* {"," *Константа*} ":" "("*Список полей*")".
Селектор варианта = [*Поле признака* ":"] *Тип признака*.
Тип признака = *Имя ординального типа*.
Поле признака = *Имя*.

Константа, предшествующая варианту, должна обозначать значение, относящееся к типу признака. В данной вариантной части каждое такое значение должно встречаться точно один раз. Если в селекторе варианта встречается поле признака, то его имя считается именем поля, принадлежащего к типу признака.

В любой данный момент *активным* может быть лишь один из вариантов данной вариантной части. Если есть поле признака, то активным будет вариант, которому предшествует значение, равное значению поля признака.Если поля признака нет, то активным будет тот вариант, к которому относилась самая последняя по обращению компонента.

Значение списка полей определяется значением каждого из полей, заданных в фиксированной части, и значением вариантной части. Значение вариантной части состоит из указания (indication) того, какой вариант активен, значения поля признака (если он есть) и значения активного варианта.

Примеры записных типов с вариантными частями.

```
record
case NameKnown: Boolean of
   false: ( );
   true:  (Name: packed array [1..NameMax] of Char)
end
```

```
record
   X, Y: Real;
   Area: Real;
case S: Shape of
   Triangle:  ( Side: Real;
                Inclination, Angle1, Angle2: Angle
              );
   Rectangle: ( Side1, Side2: Real;
                Skew, Angle3: Angle
              );
   Circle:    ( Diameter: Real )
end
```

6.2.3. Множественные типы. Множественный тип определяется как множество значений множества-степени, построенного на основе значений *базового типа.* Таким образом, каждое значение множественного типа есть множество, содержащее нуль или более компонент (элементов), причем каждый элемент — значение базового типа.

Множественный тип = "set" "of" *Базовый тип.*
Базовый тип = *Ординальный тип.*

Примеры множественных типов:

```
set of Char
packed set of 0..11
```

6.2.4. Файловые типы. Любой файловый тип выглядит как объединение последовательности компонент, имеющих один-единственный тип (тип компонент), некоторого положения в этой последовательности (позиции) и режима (mode), указывающего, формируется файл или просматривается. Число компонент в последовательности называется *длиной* файла; при описании файлового типа его длина не фиксируется. Файл называется *пустым,* если его длина равна нулю.

Файловый тип = "file" "of" *Тип компоненты.*

Тип компоненты любого файлового типа должен быть присваиваемым (см. разд. 6.5). Файл, находящийся в режиме *просмотра* (inspection), может быть установлен на любую компоненту последовательности или в положение *конца файла* (end-of-file). Файл, находящийся в режиме *формирования* (generation), постоянно установлен в положение конца файла. Работа с файловыми значениями идет с помощью предописанных процедур и функций обработки файлов (см. разд. 11).

Имя предопределенного составного типа Text относится к особому файловому типу, в котором последовательность (символов.— *Примеч. пер.*) состоит из нуля или более строчек (lines)*. Любая строчка в свою очередь состоит из нуля или более символов (значений типа Char), за которыми следует особый маркер *конца строки* (end-of-line). Переменные типа Text называются *текстовыми файлами* (text-file). Если непустой текстовый файл находится в режиме просмотра, то перед положением конца файла всегда находится конец строки. Для работы с текстовыми файлами существует несколько дополнительных, предописанных процедур и функций (см. разд. 11.5 и 12).

---

* Следуя определенной традиции, мы сохраняем одинаковый перевод для терминов line и string — «строка». Надеемся, что «контекстные» связи позволяют проводить естественное отождествление. Там же, где это сделать невозможно, для термина line будет употребляться уточняющее — «строчка». — *Примеч. пер.*

6.3. ССЫЛОЧНЫЕ ТИПЫ

Ссылочный тип отличается от составного и простого типов тем, что его множество значений — *динамическое,* т. е. значения любого ссылочного типа порождаются и уничтожаются в процессе выполнения программы. Множество значений всякого ссылочного типа всегда содержит некоторое особое значение, представляемое с помощью слова nil. Каждое из значений этого множества должно в программе порождаться с помощью предописанной процедуры New (см. разд. 11.4.2). Такие значения называются *идентифицирующими* (identifying values), поскольку каждое из них идентифицирует некоторую переменную; ее будем называть *идентифицированной переменной* (identified variable)* (см. разд. 7.3). Идентифицированная переменная относится к *типу области* (domain type) соответствующего ссылочного типа. Идентифицирующее значение и его идентифицированная переменная могут уничтожаться с помощью предописанной процедуры Dispose (см. разд. 11.4.2). При окончании программы все идентифицирующие значения, порожденные в ней, перестают существовать.

Ссылочный тип = "↑" *Тип области*| *Имя ссылочного типа.*
Тип области = *Имя типа.*
Имя ссылочного типа = *Имя типа.*

6.4. ПРИМЕР РАЗДЕЛА ОПРЕДЕЛЕНИЯ ТИПОВ

```
type
   Natural = 0..Maxint;
   Color = (Red, Yellow, Green, Blue);
   Hue = set of Color;
   Shape = (Triangle, Rectangle, Circle);
   Year = 1900..2100;
```

---

* Наличие такого понятия еще раз подтверждает необходимость введения нового термина в языки программирования или хотя бы нового перевода для слова «identifer». Русское толкование слов «идентификация», «идентификатор» подразумевает однозначное определение объекта (в общем случае, без контекстных зависимостей). Поэтому, как только идентификатор связывается с понятием «область определения», «контекст», он сразу же теряет специфические черты идентификатора (глобальной однозначности) и превращается в контекстозависимое имя. Ссылочные же значения (в машинной интерпретации — «адреса») идентифицируют переменную, а точнее значение, совершенно однозначно. Поэтому в книге и появляются идентифицирующие значения и идентифицированные переменные. Однако интересно было бы узнать, как авторы стали бы называть переменные, идентифицированные с помощью идентификаторов, т. е. того, что мы в книге (переводе) называем «именем»? — *Примеч. пер.*

```
Card = array [1..80] of Char;
String18 = packed array [1..18] of Char;
Complex = record Re, Im: Real end;
PersonPointer = ↑ Person;
Relationship = (Married, Coupled, Single);
Person =
   record
      Name, Firstname: String18;
      BirthYear: Year;
      Sex: (Male, Female);
      Father, Mother: PersonPointer;
      Friends, Children: file of PersonPointer;
      ExRelationshipCount: Natural;
   case Status: Relationship of
      Married, Coupled: (SignificantOther: PersonPointer);
      Single: ( )
   end;
MatrixIndex = 1..N;
SquareMatrix = array [MatrixIndex, MatrixIndex] of Real;
```

6.5. СОВМЕСТИМОСТЬ ТИПОВ

Два типа называются совместимыми, если справедливо любое из четырех следующих условий:

а) они представляют собою один и тот же тип;

б) один представляет собою диапазон другого, или оба они — диапазоны некоторого исходного* (host) типа;

в) оба типа — множественные, причем их базовые типы — совместимы, и оба они либо упакованы, либо нет;

г) оба типа — строковые, причем у них одинаковое число компонент**.

---

* Это отнюдь не означает, что термин «host» всегда следует переводить как «исходный». — *Примеч. пер.*

** Типы имеют компоненты? Вероятно, речь здесь идет о компонентах строки, но тем не менее так говорить не стоит. Дело в том, что «аксиоматическое определение» Паскаля, данное в свое время Хоаром, и вообще «математический» подход к понятию «тип», за который «ратует» Н. Вирт, исходят из того, что «тип» — это множества значений, которые можно принимать... и т. д. Значит элементы этого множества и есть компоненты типа. Это еще раз подчеркивает, что «вербальные» определения во многих случаях лишь затуманивают суть дела. — *Примеч. пер.*

Типы называются *присваиваемыми* (assignable), если они не файловые, или другие составные типы с компонентами неприсваиваемого типа.

Значение, относящееся (possessing) к типу T2, называется *совместимым по присваиванию* (assignment-compatible) с типом T1, если справедливо любое из четырех следующих условий:

а) T1 и T2 — представляют собой один и тот же присваиваемый тип;

б) T1 — Real, а T2 — Integer;

в) T1 и T2 — совместимые ординальные типы или совместимые множественные типы, а значение (упомянутое выше. — *Примеч. пер.*) — элемент из множества значений, определяемых типом T1;

г) T1 и T2 — совместимые строковые типы.

Ситуация, когда требуется совместимость по присваиванию, причем T1 и T2 — оба совместимые ординальные типы или совместимые множественные типы, а значение не является элементом множества значений, определенного типом T1, считается ошибкой.

7. ПЕРЕМЕННЫЕ

Переменная относится к типу, определяемому ее описанием, и может принимать значения только этого типа.

Переменная *не определена* (undefined), если она не имеет значения своего типа. Переменная называется *полностью неопределенной* (totally undefined), если она не определена и, более того, если полностью не определена каждая компонента такой (составной) переменной. При порождении каждая переменная полностью не определена. Любая переменная, описанная в некотором блоке, порождается при активации блока и уничтожается по окончании активации (см. разд. 10). Каждая идентифицированная переменная порождается или уничтожается с помощью предописанных процедур New или Dispose (см. разд. 6.3 и 11.4).

Описание переменных вводит одно или более имен переменной и тип, к которому относится каждое из них. Описания переменных собираются вместе и образуют раздел описания переменных.

Раздел описания переменных = ["var" *Описание переменных* ";" {*Описание переменных* ";"}].
Описание переменных = *Список имен* ":" *Тип*.
Имя переменной = *Имя*.

Пример раздела описания переменных:

```
var
   W, X, Y: Real;
   Z: Complex;
   I, J: Integer;
   K: 0..9;
   P, Q: Boolean;
   Operator: (Plus, Minus, ...);
   GrayScale: array [0..63] of Real;
   VideoPotential: array [Color, Boolean] of Complex;
   Light: Color;
   F: file of Char;
   Hue1, Hue2: set of Hue;
   P1, P2: PersonPointer;
   A, B, C: SquareMatrix;
   Minneapolis, Zuerich: packed record
                            Area: Real;
                            Population: Natural;
                            Capital: Boolean
                         end;
```

В РБНФ обращение к переменной обозначается метаименем *Переменная.*

Переменная = *Полная переменная* | *Переменная-компонента* | *Идентифицированная переменная* | *Буферная переменная*

7.1. ПОЛНЫЕ ПЕРЕМЕННЫЕ

Любая полная переменная представляет собою переменную, обозначенную именем переменной.

Полная переменная = *Имя переменной.*

Примеры полных переменных:

```
Input
P1
VideoPotential
```

7.2. ПЕРЕМЕННЫЕ-КОМПОНЕНТЫ

Компонента любой составной переменной — это также некоторая переменная; переменная-компонента — средство обращения

к компоненте составной переменной. Синтаксис переменной-компоненты зависит от типа составной переменной.

Переменная-компонента = *Индексированная переменная* | *Обозначение поля*.

Доступ или указание на компоненту любой составной переменной предполагает доступ или указание на эту составную переменную.

7.2.1. Индексированные переменные. Индексированная переменная представляет собою компоненту переменной-массива. Переменная-массив — это переменная, относящаяся к массивовому типу.

Индексированная переменная = *Переменная-массив* "["*Индекс* {","
Индекс }"]".
Индекс = *Ординальное выражение*.
Переменная-массив = *Переменная*.

Компонента, к которой идет обращение, — компонента, соответствующая значению индексирующего выражения. Это значение при обращении должно быть совместимо по присваиванию (см. разд. 6.5) с типом индекса. Если индексных выражений несколько, то порядок их вычислений зависит от реализации

Примеры:

```
GrayScale[12]
GrayScale[I+J]
VideoPotential[Red, True]
```

Если встречается более одного индекса, как, например, в

VideoPotential [Red, True]

то это просто сокращение для такой записи:

VideoPotential [Red] [True].

7.2.2. Обозначение поля. Обозначение поля относится к некоторому полю переменной-записи. Переменная-запись — это переменная, относящаяся к записному типу.

Обозначение поля = [*Переменная-запись* "."] *Имя поля*.
Переменная-запись = *Переменная*.

Обозначаемое поле есть поле, соответствующее имени поля; можно употреблять лишь имена полей, указанные в записном типе этой переменной-записи. Внутри оператора присоединения, перечисляющего переменные-записи (см. разд. 9.2.4), можно опускать переменную-запись (точнее, ее имя. — *Примеч. пер.)* и символ ".".

Примеры обозначений поля:

```
Z.Re
VideoPotential[Red,True].Im
P2↑.Mother
```

Если некоторый вариант переменной-записи становится неактивным, то все компоненты этого варианта становятся полностью неопределенными. Если у вариантной части нет поля признака, то любое обращение (доступ) к компоненте какого-либо варианта делает этот вариант активным, а другие — неактивными. Если существует обращение или ссылка на любую из компонент варианта, который становится неактивным или уже неактивен, то это считается ошибкой. В случае неопределенности поля признака ни один из вариантов не будет активным. Никакое поле признака не должно фигурировать в качестве фактического параметра-переменной.

7.3. ИДЕНТИФИЦИРОВАННЫЕ ПЕРЕМЕННЫЕ

Идентифицированная переменная обозначает переменную, идентифицированную значением переменной-ссылки. Переменная-ссылка — это переменная, относящаяся к ссылочному типу.

Идентифицированная переменная = Ссылочная переменная "↑".
Ссылочная переменная = Переменная.

Любое обращение к идентифицированной переменной предполагает обращение к ссылочной переменной (переменной-ссылке); если такая переменная в этот момент не определена или имеет значение nil, то это считается ошибкой. Ошибкой также считается и уничтожение идентифицирующего значения, если переменная, которую это значение идентифицирует, продолжает существовать*.

Примеры идентифицированных переменных:

```
p1↑
p1↑.Father↑
p1↑.Friends↑↑
```

* Эта ситуация встречается, если переменной-ссылке, получившей значение с помощью процедуры New, присваивается другое значение без «предварительного» обращения к Dispose, т. е. ссылка на порожденную переменную теряется, но сама она, хотя к ней уже нельзя обратиться, продолжает существовать. — *Примеч. пер.*

7.4. БУФЕРНЫЕ ПЕРЕМЕННЫЕ

Переменная-файл (файловая переменная) — это переменная, относящаяся к файловому типу. С каждой переменной-файлом связана так называемая буферная переменная.

Буферная переменная = *Переменная-файл* "↑".
Переменная-файл = *Переменная.*

Если переменная-файл относится к типу Text, то соответствующая буферная переменная относится к типу Char, в противном же случае — к типу компонент упомянутого файлового типа, т. е. типа, к которому относится переменная-файл. Буферная переменная используется для доступа к текущей компоненте переменной-файла. Если существуют ссылки на буферную переменную, то изменять последовательность (sequence), положение (position) или режим (mode) переменной-файла нельзя, это ошибка. Любое обращение (к) или ссылка на буферную переменную предполагает обращение или ссылку на соответствующую переменную-файл.

В разд. 11.4, 11.5 и 12 приводятся предописанные процедуры и функции, манипулирующие переменными-файлами.

Если для любого текстового файла F справедлив предикат (функция) eoln(F) (см. разд. 11.5.2), то буферная переменная F↑ принимает значение символа «пробел» (''). Таким образом, обнаружить признак конца строчки в F можно только с помощью eoln(F).

Примеры буферных переменных:

```
Input↑
Pl↑.Friends↑
Pl↑.Friends↑↑.Children↑
```

8. ВЫРАЖЕНИЯ

Любое *выражение* обозначает правило вычислений, дающих при выполнении выражения некоторое значение. Исключением является лишь случай, когда выражение активирует некоторую функцию, а та заканчивается оператором перехода (см. разд. 9.1.3 и 10). Значение выражения зависит от значений констант, границ и переменных выражения и от операций и функций, включенных в выражение.

Выражение = *Простое выражение* [*Операция отношения Простое выражение*].
Простое выражение = [*Знак*] *Терм* {*Аддитивная операция Терм*}.
Терм = *Фактор* {*Мультипликативная операция Фактор*}.
Фактор = *Константа без знака*| *Имя границ*| *Переменная*|
Конструктор множества| *Обозначение функции*|
"not" *Фактор*| "("*Выражение*")".
Константа без знака = *Число без знака*|*Строка символов*|
Имя константы|"nil".
Конструктор множества = "[" [*Описание элемента* {"," *Описание элемента*}]"]".
Описание элемента = *Ординальное выражение* [".." *Ординальное выражение*].
Обозначение функции = *Имя функции* [*Список фактических параметров*].
Операция отношения = "=" | "<>" | "<" | "<=" | ">" | ">=" | "in"
Аддитивная операция = "+" | "—" | "or".
Мультипликативная операция = "*" | "/" | "div" | "mod" | "and".

Ординальные выражения — это выражения, относящиеся к ординальному типу. Логическое или целое выражение — это ординальное выражение, относящееся соответственно к типу Boolean или Integer.

8.1. ОПЕРАНДЫ

Любая мультипликативная операция в терме имеет два операнда: один — та часть терма, которая предшествует операции, второй — фактор, идущий сразу же за операцией. Аддитивная операция в простом выражении имеет также два операнда: один — та часть простого выражения, которая предшествует операции, второй — терм, идущий сразу же за операцией. Два операнда любой операции отношения — простые выражения: предшествующее и следующее за самой операцией. Операндом для (одиночного) знака является простое выражение из терма, непосредственно следующего за знаком. Операндом для not в некотором факторе является фактор, идущий следом за not.

Порядок вычисления операндов любой операции зависит от реализации. Всякая программа, удовлетворяющая стандарту, не должна ориентироваться на какой-либо порядок. Левый операнд может вычисляться и до, и после правого операнда, они могут вычисляться даже одновременно. Иногда, при определенных значениях других операндов, некоторые из операндов могут вообще не вычисляться. Например, вычисление выражения (j * (i div j) при нулевом значении j в одной реализации может дать нулевое значение, а в другой приведет к ошибке, вызванной делением на нуль.

Тип любого фактора выводится (is derived) на основании типов, его составляющих (т. е. переменных или функций). Если тип составляющей — диапазон, то тип фактора — исходный для этого диапазона тип. Если же тип составляющей — множественный, причем его базовым типом является некоторый диапазон, то тип фактора — множественный тип, базовым типом которого будет тип основания упомянутого диапазона; во всех других случаях тип фактора — тот же, что и тип его составляющей.

Имя nil относится к любому ссылочному типу и ему соответствует значение nil.

Конструктор множества обозначает значение множества. Если в конструкторе нет никаких описаний элементов, то он обозначает пустое множество; такое множество — одно из значений любого множественного типа. Во всех других случаях элементы значения множества описываются с помощью описаний элементов в конструкторе множества. Все выражения из описаний элементов в конструкторе множества должны быть одинакового типа, он и будет базовым типом для типа данного конструктора множества. Тип конструктора множества одновременно и упакованный, и неупакованный, он совместим с любым другим множественным типом, имеющим совместимый базовый тип.

Всякое описание элемента множества, состоящее из единственного выражения, описывает элемент со значением, обозначенным (denoted) этим выражением. Описание элемента, имеющее вид a..в, описывает элемент, каждое из значений которого (x) удовлетворяет отношению a <= x <= в. Если a > в, то a...в не обозначает никаких элементов. Порядок вычисления выражений в описании элемента и порядок вычисления самих описаний элементов в конструкторе множества зависит от реализации.

Вычисление фактора, содержащего переменную, задает обращение к этой переменной и обозначает ее значение; если переменная не определена, то это ошибка.

Вычисление фактора, содержащего обозначение функции, задает активацию функции, указанной именем функции (см. разд. 10.3). В качестве ее формальных параметров (см. разд. 11.3) подставляются соответствующие фактические параметры. После окончания активации алгоритма фактор обозначает значение результата активации; если результат не определен, то это ошибка.

8.2. ОПЕРАЦИИ

Правила композиции задают *приоритеты* операций в соответствии с их разбиением на четыре класса. Наивысший приоритет — у операции not, затем идут так называемые мультипликативные

операции, следом — аддитивные операции и, наконец, с наименьшим приоритетом — операции отношения. Последовательности операций с одинаковым приоритетом выполняются слева направо. Система приоритетов находит свое отражение в правилах РБНФ для конструкций: *Выражение, Простое выражение, Терм* и *Фактор* (приводившихся выше).

В соответствии с типом операндов и результатом операции классифицируются как арифметические, логические, множественные и операции отношения.

8.2.1. Арифметические операции. Арифметические операции выполняются над целыми или вещественными операндами и порождают целый или вещественный результат.

Ниже суммированы операции с одним операндом, т. е. знаки.

Операция	*Действие*	*Тип операнда*	*Тип результата*
+	тождественное	Integer или Real	тип операнда
—	изменение знака	Integer или Real	тип операнда

В следующей же таблице суммированы операции с двумя операндами.

Операция	*Действие*	*Тип операнда*	*Тип результата*
+	сложение	Integer или Real	Integer или Real
—	вычитание	Integer или Real	Integer или Real
*	умножение	Integer или Real	Integer или Real
/	деление	Integer или Real	Real
div	деление	Integer	Integer
mod	остаток	Integer	Integer

Результат операции сложения, вычитания и умножения будет целого типа, если оба операнда — целого типа; в других случаях результат — вещественного типа.

Вычисление терма вида x/y будет ошибкой, если y равен нулю.

Вычисление терма вида x div y будет ошибкой, если y равен нулю. Во всех других случаях значение терма удовлетворяет следующим двум правилам:

а) abs(x) — abs(y) < abs((x div y) * y) <= abs(x) и
б) x div y = 0, если abs(x) < abs(y);

в других случаях x div y будет положительным, если x и y имеют один и тот же знак, и отрицательным, если x и y — с разными знаками.

Вычисление терма вида x mod y будет ошибкой, если y меньше или равно нулю; в других случаях существует k, при котором x mod y удовлетворяет следующему отношению:

$$0 <= x \bmod y = x - k * y < y.$$

Если в случае целочисленных операций оба операнда лежат в диапазоне — Maxint .. Maxint и верный результат находится в том же диапазоне, то стандартная реализация должна давать правильный результат. Если же операнды или результат не находятся в диапазоне — Maxint .. Maxint, то при реализации можно выбирать: выполнять ли операцию корректно или рассматривать ее как ошибку.

Операции и предописанные функции (см. разд. 11.5), дающие вещественный результат, всегда следует считать приближенными, а не точными. Точность таких вещественных операций и предописанных функций зависит от реализации.

8.2.2. Логические операции. Логические операции приводятся в следующей таблице:

Операция	*Действие*	*Тип операндов*	*Тип результата*
not	логическое «не»	Boolean	Boolean
and	логическое «и»	Boolean	Boolean
or	логическое «или»	Boolean	Boolean

8.2.3. Операции над множествами. Операции над множествами суммированы в приведенной ниже таблице. Оба операнда должны всегда относиться к совместимым типам (см. разд. 6.5). Тип результата будет упакованным, если упакованы типы обоих операндов. Тип результата не упакован, если типы обоих операндов не упакованы.

Операция	*Действие*	*Тип операндов*	*Тип результата*
+	объединение множеств	set of T	set of T
−	разность множеств	set of T	set of T
*	пересечение множеств	set of T	set of T

8.2.4. Операции отношения. Операции отношения приведены ниже. За исключением операции in типы операндов либо должны быть совместимыми, либо один должен быть Real, а другой — Integer. Для операции in первый (левый) операнд должен отно-

ситься к ординальному типу, совместимому с базовым типом того множественного типа, к которому относится второй операнд.

Выражение x <= y (где x и y — множества) дает значение истина, если каждый элемент x является элементом y, т. е. если x есть подмножество y. Упорядоченность совместимых строк определяется упорядоченностью значений типа Char (см. разд. 6.1.2).

Операция	*Действие*	*Тип операндов*	*Тип результата*
=	равенство	простой, ссылочный, множественный, строковый	Boolean
<>	неравенство	простой, ссылочный множественный, строковый	Boolean
<=	меньше или равно	простой, строковый	Boolean
<=	включение множеств	множественный	Boolean
>=	больше или равно	простой, строковый	Boolean
>=	включение множеств	множественный	Boolean
<	меньше	простой, строковый	Boolean
>	больше	простой, строковый	Boolean
in	принадлежность множеству	ординальный и множественный	Boolean

Примеры факторов:

```
X
15
(W + X + Y)
sin(X+Y)
[Red, Light, Green]
[1, 5, 10..19, 60]
not P
```

Примеры термов:

```
X * Y
I/(1-I)
Q and not P
(X <= Y) and (Y < W)
```

Примеры простых выражений:

```
X + GrayScale[2 * I]
-X
P or Q
Hue1 + Hue2
I*J + 1
```

Примеры выражений:

```
X = 1.5
P <= Q
(I < J) = (J < K)
Light in Hue1
```

9. ОПЕРАТОРЫ

Операторы обозначают алгоритмические действия; про них говорят, что они *выполняемые* (executable). Перед любым оператором допускается метка, на которую можно ссылаться с помощью оператора перехода. Операторы объединяются в раздел операторов:

Простой оператор = *Пустой оператор*| *Оператор присваивания*|
Оператор процедуры| *Оператор перехода*.
Пустой оператор = .

9.1. ПРОСТЫЕ ОПЕРАТОРЫ

Простым называется оператор, в который не входят как составные части другие операторы. Пустой оператор не содержит никаких символов и не обозначает никаких действий.

Оператор = [*Метка* ":"] (*Простой оператор*|*Сложный оператор*).
Раздел операторов = *Составной оператор*.

9.1.1. Оператор присваивания. Оператор присваивания предназначен для обращения к переменной или результату активации функции и замене их текущего значения на значение, полученное при вычислении выражения.

Оператор присваивания = (*Переменная*| *Имя функции*) ":="
Выражение.

Значение выражения должно быть совместимо по присваиванию (см. разд. 6.5) с типом переменной или типом имени функции. Порядок обращения к переменной или результату и вычисления выражения зависит от реализации. Обращение к переменной формирует ссылку на переменную, которая существует до тех пор, пока не пройдет присваивание значения.

Примеры операторов присваивания:

```
X := Y + GrayScale[31]
P := (1 <= I) and (I < 100)
I := sqr(K) - (I*J)
Hue2 := [Blue, succ(C)]
```

9.1.2. Операторы процедуры. Оператор процедуры предназначен для активации процедуры, обозначенной именем процедуры. Оператор процедуры может содержать список фактических параметров, которые подставляются вместо формальных параметров, определенных в описании процедуры (см. разд. 11.1).

Оператор процедуры = *Имя процедуры* [*Список фактических параметров*| *Список параметров вывода*].

Если имя процедуры именует стандартную процедуру Write или Writeln, то фактические параметры должны следовать синтаксису, указанному для конструкции *Список параметров вывода*. Если же имя процедуры именует любую другую предописанную процедуру, то фактические параметры должны удовлетворять правилам, приведенным в разд. 11.4 и 12.

Примеры операторов процедур:

```
Next
Transpose(A,N,N)
Bisect(Fct, -1.0, +1.0, X)
Writeln(Output, ' Title' )
```

9.1.3. Операторы перехода. Оператор перехода предназначен для указания, что процесс выполнения должен продолжаться с другого «места» программы, а именно с «точки» программы (programm-point), обозначенной меткой (см. разд. 10.1 и 10.3).

Оператор перехода = "goto" *Метка*.

Оператор, перед которым стоит некоторая метка, и любой оператор перехода, ссылающийся на эту метку, должны удовлетворять одному из двух следующих правил:

а) оператор либо должен содержать оператор перехода, либо

должен быть одним из операторов в последовательности операторов (см. разд. 9.2), содержащей оператор перехода;

б) оператор должен быть одним из операторов в последовательности операторов из составного оператора, входящего в раздел операторов того блока, где описана метка. Оператор же перехода должен находиться в разделе описания процедур и функций указанного блока (см. разд. 10.1).

Смысл этих правил в том, что они запрещают операторы перехода, передающие управление извне в сложный оператор и в процедуру или функцию. Первое правило к тому же запрещает передачу управления между «ветвями» условного оператора.

Если оператор перехода и соответствующая «точка» программы не находятся в одном разделе операторов, то каждая активация, не удовлетворяющая какому-либо из двух следующих условий, заканчивается (см. разд. 10.3):

а) точка программы входит в данную активацию;

б) активация содержит «точку» активации другой активации, которая не заканчивается (т. е. удовлетворяет одному из этих двух условий).

9.2. СЛОЖНЫЕ ОПЕРАТОРЫ

Сложные операторы — это конструкции, состоящие из других операторов, причем эти операторы либо выполняются последовательно (составные операторы), либо — в зависимости от условия (выбирающие операторы), либо повторяются (циклические операторы), либо выполняются в некоторой расширенной области действия (scope) (операторы присоединения).

Сложный оператор = *Составной оператор* | *Выбирающий оператор* | *Циклический оператор* | *Оператор присоединения*.

Последовательность операторов представляет собою перечисление операторов, которые должны выполняться в порядке написания, если только оператор перехода не предпишет что-то другое.

Последовательность операторов = *Оператор* {";" *Оператор*}.

Последовательность операторов используется в составных операторах (см. разд. 9.2.1) и циклических операторах (см. разд. 9.2.3.2).

9.2.1. Составные операторы. Составной оператор задает выполнение последовательности операторов. Слова begin и end выполняют роль операторных скобок.

Составной оператор = "begin" *Последовательность операторов* "end".

Примеры составных операторов:

```
begin   end
begin   W := X; X := Y; Y := W  end
```

9.2.2. Выбирающие операторы. Выбирающий оператор назначает для выполнения один из составляющих его операторов.

Выбирающий оператор = *Условный оператор*| *Оператор варианта.*

9.2.2.1. Условные операторы. Условный оператор указывает, что следующий за словом then оператор должен быть выполнен только в том случае, если логическое выражение дает значение «истина». Если же оно дает значение «ложь», то выполняется оператор, следующий за словом else, если он вообще есть.

Условный оператор = "if" *Логическое выражение* "then" *Оператор* ["else" *Оператор*].

Замечание. Синтаксическая двусмысленность конструкции:

if e1 then if e2 then s1 else s2

разрешается такой ее интерпретацией: считается, что она эквивалентна конструкции:

```
if e1 then
   begin if e2 then s1 else s2 end
```

Примеры условных операторов:

```
if X < 1.5 then W := X + Y else W := 1.5
if P1 <> nil then P1 := P11.Father
```

9.2.2.2. Операторы варианта. Оператор варианта содержит ординальное выражение (индекс варианта) и список операторов, перед каждым из которых стоит одна или несколько констант, относящихся к типу индекса варианта. Оператор указывает, что необходимо выполнить оператор, перед которым стоит значение индекса варианта.

Если ни перед одним оператором не стоит константа, равная этому значению, то это ошибка. Каждому значению должна соответствовать, самое большое, одна константа варианта.

Оператор варианта = "case" *Индекс варианта* "of" *Вариант* {";"*Вариант*} [";"] "end".
Индекс варианта = *Ординальное выражение*.
Вариант = *Константа* {"," *Константа*} ":" *Оператор*.

Примеры операторов варианта:

```
case Operator of
   Plus:   W := X + Y;
   Minus:  W := X - Y;
   Times:  W := X * Y
end

case I of
   1:  Y := sin(X);
   2:  Y := cos(X);
   3:  Y := exp(X);
   4:  Y := ln(X)
end

case Plf.Status of
   Married:  P1 := Plf.SignificantOther;
   Coupled.  P2 := Plf.SignificantOther;
   Single:
end
```

9.2.3. Циклические операторы. Циклические операторы указывают, что выполнение некоторых операторов необходимо повторять. Если число повторений известно заранее, т. е. до начала повторений, то в этом случае наиболее подходящей конструкцией будет цикл с шагом. В противном же случае следует использовать циклы с предусловием (со словом While) или с постусловием (со словом Repeat).

Циклические операторы = *Цикл с предусловием* | *Цикл с постусловием* | *Цикл с шагом*.

9.2.3.1. Цикл с предусловием (со словом While).

Цикл с предусловием = "while" *Логическое выражение* "do" *Оператор*.

Оператор повторно выполняется до тех пор, пока выражение не даст значение «ложь». Если уже в самом начале получено такое значение, то оператор не выполняется вовсе. Цикл с предусловием

while B do S

если только S не содержит какого-либо помеченного оператора, эквивалентен такой последовательности:

if B then begin S; while B do S end

Примеры циклов с предусловием:

```
while GrayScale[I] < X do I := succ(I)
```

```
while I > 0 do
   begin
      if odd(I) then Y:= Y * X;
      I := I div 2;
      X := sqr(X)
   end
```

```
while not eof(F) do
   begin  P(F↑); Get(F)  end
```

9.2.3.2. Циклы с постусловием (со словом Repeat).

Цикл с постусловием = "repeat" *Последовательность операторов* "until" *Логическое выражение.*

Последовательность операторов повторно выполняется до тех пор, пока выражение не даст значение «истина» (по крайней мере один раз оно выполняется). Оператор цикла с постусловием

repeat S until B

если S не содержит какого-либо помеченного оператора, эквивалентен такой последовательности:

begin S; if not B then repeat S until B end

Примеры циклов с постусловием:

```
repeat  K := I mod J;  I := J;  J := K  until J = 0
```

```
repeat
   P(F↑);
   Get(F)
until eof(F)
```

9.2.3.3. Циклы с шагом (со словом For). Цикл с шагом указывает, что необходимо повторять выполнение оператора и одновременно присваивать переменной, называемой *управляющей переменной цикла,* последовательно возрастающие значения.

Цикл с шагом = "for" *Управляющая переменная* ":=" *Начальное значение*
("to"|"downto") *Конечное значение*
"do" *Оператор*.
Управляющая переменная = *Имя переменной*.
Начальное значение = *Ординальное выражение*.
Конечное значение = *Ординальное выражение*.

}

Управляющая переменная должна быть локализована в том блоке (см. разд. 10.2), в разделе операторов которого находится данный оператор цикла с шагом. Эта переменная должна относиться к ординальному типу, совместимому с типами начального и конечного значений.

Будем говорить, что оператор S затрагивает (потенциально) переменную V, если выполняется одно из следующих условий:

а) S — оператор присваивания, присваивающий значение переменной V;

б) переменная V встречается в S в качестве фактического параметра-переменной (см. разд. 11.3.2.2);

в) S — оператор процедуры, активирующий предописанные процедуры Read или Readln, причем V — один из фактических параметров;

г) S — цикл с шагом, а V — его параметр цикла (управляющая переменная).

Никакой оператор внутри цикла с шагом не должен затрагивать параметр цикла. Более того, никакая процедура или функция, описанная как локальная в том же блоке, где описан параметр цикла, не должна содержать оператор, затрагивающий этот параметр цикла. Все эти правила означают, что повторяющийся оператор не должен изменять значение параметра цикла.

Если ввести новые переменные T1 и T2, относящиеся к тому же типу, что и переменная V, и больше нигде не употребляемые, и еще одну переменную P, относящуюся к логическому типу, то справедлива следующая эквивалентность (с исключениями, отмеченными в комментариях):

for V := e1 to e2 do S

эквивалентно

```
begin
  T1 := e1; T2 := e2;
  if T1 >= T2 then
    begin
      { T2 должен быть совместим по присваиванию с типом V }
      V := T1; P := false;
```

```
            repeat
               S;
               if V = T2 then P := true else V := pred(V)
            until P
         end
      { V не определено  }
   end
```

А оператор

for V := e1 downto e2 do S

эквивалентен

```
   begin
      T1 := e1; T2 := e2;
      if T1 <= T2 then
         begin
            { T2 должен быть совместим по присваиванию с типом V}
            V := T1; P:= false;
            repeat
               S;
               if V = T2 then P := true else V := succ(V)
            until P
         end
      { V не определено}
   end
```

Примеры операторов цикла:

```
for I := 1 to 63 do
   if GrayScale[I] > 0.5 then write ('*') else write (' ')
```

```
for I := 1 to n do
   for J := 1 to n do
      begin
         X := 0;
         for K := 1 to n do X := X + A[I,K] * B[K,J];
         C[I,J] := X
      end
```

```
for Light := Red to pred(Light) do
   if Light in Hue2 then Q(Light)
```

9.2.4. Операторы присоединения. Оператор присоединения обращается к каждой переменной-записи из его списка и устанавливает на них «указатели», а затем выполняется входящий в него оператор. «Указатели» существуют до конца выполнения этого оператора-компоненты.

Оператор присоединения = "with" *Список переменных-записей* "do" *Оператор*.
Список переменных-записей = *Переменная-запись* "," *Переменная-запись*.

Область действия (см. разд. 10.2) каждого имени поля из типа, относящегося к (единственной) переменной-записи, перечисленной в операторе присоединения, расширяется таким образом, что в нее включается оператор-компонента. Внутри этой расширенной области действия имя поля может встречаться в обозначении поля без указания переменной-записи и будет обозначать соответствующее поле переменной, на которую установлен «указатель».

Запись

with r1, r2, ..., rn do S

представляет собою лишь сокращение такой записи:

```
with r1 do
   with r1 do

         with rn do S
```

Пример оператора присоединения:

```
with Date do
   if Month = 12 then
      begin  Month := 1;  Year := succ(Year)  end
   else Month := succ(Month)
```

Он эквивалентен такому оператору:

```
if Date.Month = 12 then
   begin  Date.Month := 1;  Date.Year := succ(Date.Year)  end
else Date.Month := succ(Date.Month)
```

10. БЛОКИ, ОБЛАСТИ ДЕЙСТВИЯ И АКТИВАЦИИ

Блоки представляют собой основную «строительную» конструкцию для построения программ (см. разд. 13), процедур и функций (см. разд. 11). Правила *области действия* определяют,

где можно использовать имя введенное в некотором конкретном месте; эти правила основываются на статической (текстуальной) структуре программы. Правила же активации определяют, каким именем или меткой обозначается объект (например, переменная). Эти правила основываются на динамической структуре программы, т. е. на процессе ее выполнения.

10.1. БЛОКИ

Блок состоит из нескольких разделов определений и описаний, любой из которых может быть пустым, и одного раздела операторов.

Блок = *Раздел описания меток*
Раздел определения констант
Раздел определения типов
Раздел описания переменных
Раздел описания процедур и функций
Раздел операторов.

Раздел описания меток вводит одну или несколько меток, каждая из которых должна стоять перед каким-либо одним оператором в разделе операторов.

Раздел описания меток = ["label" *Последовательность цифр*
{"," *Последовательность цифр*} ";"].
Метка = *Последовательность цифр.*

Значение (spelling) метки — целое значение, описываемое последовательностью цифр по правилам обычной десятичной нотации; это значение не должно превышать 9999.

10.2. ОБЛАСТЬ ДЕЙСТВИЯ

Любое определение или описание вводит написание* (spelling) некоторых имен или меток и связывает с этим написанием какой-либо специфический смысл (например, имя переменной). Та часть программы, в которой под именем должно пониматься именно

* В стандарте ИСО такой термин отсутствует. Похоже, что его введение продиктовано желанием авторов иметь одно название и для имен, и для меток. Далее, если это не будет приводить к неточностям, мы будем все же использовать термин «имя». – *Примеч. пер.*

это значение, называется *областью действия* соответствующего введения (introduction) — определения или описания. Каждому появлению имени внутри области действия должно обязательно предшествовать соответствующее введение. Есть только одно исключение — некоторое имя типа может появиться в качестве типа области в ссылочном типе (см. разд. 6.3) в любом месте раздела определения типов, содержащего введение этого имени.

Каждое введение, как описывается ниже, действует внутри некоторой части программы. Область действия введения — это область программы, меньшая, чем любая вложенная область, в которой действует другое введение того же самого имени.

Для блока, в котором встречается введение, действуют следующие введения: метки из раздела описания меток, имена констант из раздела определения констант или из каких-либо перечисляемых типов, имена типов из раздела определения типов, имена переменных из раздела описания переменных, имена процедур из описаний процедур (см. разд. 11.1), имена функций из описаний функций (см. разд. 11.2). Про такие метки и имена говорят, что они *локализованы* в данном блоке (локальны).

В области, окружающей любую программу, действует неявное введение стандартных предопределенных и предописанных имен.

Введение имени поля в записном типе действует в любой из следующих областей:

а) в самом записном типе;

б) в составляющем операторе из оператора присоединения, если переменная-запись оператора присоединения относится к данному записному типу;

в) в части имени поля из обозначения поля, если часть переменной-записи относится к данному записному типу. В последнем случае часть имени поля исключается из всех объемлющих областей действия.

Введение в списке параметров любого имени параметра (см. разд. 11.3.1) действует внутри списка параметров. Более того, если список параметров находится в заголовке процедуры из описания процедуры или в заголовке функции из описания функции, то вводятся и становятся действительными в блоке из описания функции или процедуры в соответствии с именами параметров имена переменных, имена границ, имена процедур или имена функций.

10.3. АКТИВАЦИИ

Активация программы (см. разд. 13) и программы или функции (см. разд. 11) представляет собою активацию блока этой программы, процедуры или функции.

Говорится, что в активацию *входят* следующие объекты (сущности), которые существуют до тех пор, пока активация не закончится:

а) *Алгоритм* – он определяется разделом операторов соответствующего блока; алгоритм начинает выполняться при активации блока, а при его окончании заканчивается и активация. (Активация может закончиться и через оператор перехода – см. разд. 9.1.3.)

б) *Точка программы* (алгоритма) соответствует метке, стоящей перед оператором в разделе операторов блока. Любое появление этой метки в операторе перехода внутри активации обозначает данную точку программы.

в) *Переменная,* соответствующая каждому имени переменной, локальному в данном блоке. В начале выполнения алгоритма переменная полностью не определена, если только имя переменной не параметр программы. Каждое появление внутри активации данного имени переменной обозначает эту переменную.

г) *Процедура,* соответствующая каждому имени процедуры, локальному в данном блоке. Процедура состоит из блока и формальных параметров из описания процедуры, которое вводит имя процедуры. Каждое появление имени процедуры внутри активации обозначает эту процедуру.

д) *Функция,* соответствующая каждому имени функции, локальному в данном блоке. Функция состоит из блока, формальных параметров и типа результата из описания функции, которое вводит имя процедуры. Каждое появление этого имени функции внутри активации обозначает эту функцию.

е) *Переменная,* соответствующая каждому имени переменной из списка имен формальных параметров-значений для данного блока. В начале выполнения алгоритма эта переменная имеет значение соответствующего фактического параметра из оператора процедуры или обозначения функции, активировавших данную процедуру или функцию. Каждое появление этого имени переменной внутри активации обозначает такую переменную.

ж) *Ссылка,* соответствующая каждому имени переменной из списка имен формальных параметров-переменных в данном блоке; эта ссылка указывает на переменную, которая в начале выполнения алгоритма обозначается соответствующим фактическим параметром. Каждое появление этого имени переменной внутри активации обозначает указанную переменную.

з) *Ссылка* на процедуру или функцию, соответствующая каждому имени из списка имен формальных параметров-процедур или параметров-функций в данном блоке. Ссылка в начале выполнения алгоритма указывает на процедуру или функцию,

которая обозначается соответствующим фактическим параметром. Каждое появление внутри активации этого имени процедуры или имени функции обозначает указанную процедуру или функцию.

и) Если активированный блок есть блок функции, то в начале выполнения алгоритма *результат* не определен.

Считается, что любая активация блока некоторой процедуры или функции должна проводиться *внутри* активации, содержащей данную процедуру или функцию. Если активация А проводится внутри активации В, то говорят, что она проводится и внутри другой активации, содержащей активацию В.

Оператор процедуры или обозначение функции, содержащиеся в некотором алгоритме и задающие активацию некоторого блока, называется для данной активации *точкой активации*.

11. ПРОЦЕДУРЫ И ФУНКЦИИ

Процедура и функция — поименованная часть программы, активируемая с помощью оператора процедуры (разд. 9.1.2) или соответственно обозначения функции (разд. 8.1). Если необходимо, то программист может описать новые процедуры или функции. Описания процедур и описания функций объединяются вместе и образуют раздел описания процедур и функций.

Раздел описания процедур и функций =
{(*Описание процедуры*|*Описание функции*) ";"}.

Кроме того, в каждой реализации предусматривается некоторое число «предописанных» процедур и функций. Поскольку они, как и все аналогичные объекты, предполагаются описанными в области действия, окружающей всю программу, то в случае описаний, переопределяющих те же имена внутри программы, никаких конфликтов не возникает.

11.1. ОПИСАНИЯ ПРОЦЕДУР

Описание процедуры предназначено для введения имени процедуры, сопоставления с этим именем некоторого блока и, возможно, списка формальных параметров. Имя процедуры и список формальных параметров вводятся с помощью заголовка процедуры из описания процедуры.

Процедура может быть описана с помощью одного-единственного описания процедуры, состоящего из заголовка процедуры и блока. Это наиболее распространенный способ.

Однако есть и другой способ — процедуру можно описывать с помощью «опережающего описания»: первое описание процедуры состоит из заголовка процедуры и директивы forward, а второе описание, находящееся в том же разделе описания процедур и функций, лишь идентифицирует процедуру и содержит ее блок. Процедура должна идентифицироваться тем же именем процедуры, которое было введено в первом описании. Заметим, что список формальных параметров (если он есть) во втором описании не указывается.

Описание процедуры = *Заголовок процедуры* ";" *Блок*|
Заголовок процедуры ";" *Директива*|
Идентификация процедуры ";" *Блок*.
Заголовок процедуры = "procedure' *Имя* [*Список формальных параметров*].
Идентификация процедуры = "procedure' *Имя процедуры*.
Имя процедуры = *Имя*.

Употребление имени процедуры в операторе процедуры внутри блока ее собственного описания предполагает рекурсивное использование этой процедуры.

Пример раздела описания процедур и функций, содержащих процедуры:

```
procedure ReadInteger (var F: Text;  var X: Integer);
   var S: Natural;
begin
   while F↑ <> ' ' do Get(F);
   S := 0;
   while F↑ in ['0'..'9'] do
       begin
         S := 10 * S + (ord(F↑)-ord('0'));
         Get(F)
       end;
   X := S
end { ReadInteger } ;

procedure Bisect(function F(X: Real): Real; A, B: Real; var Z: Real);
   var M: Real;
```

```
begin
   { пусть F(A) < 0  и  F(B) > 0 }
   while abs(A-N) > 1e-10 * abs(A) do
      begin
         M := (A + B) / 2.0;
         if F(M) < 0 then  A := M  else  B := M
      end;
   Z := M
end { Bisect } ;

procedure GCD(M, N: Integer;  var X, Y, Z: Integer);
   {Наибольший общий делитель X для M и N, причем M>=0  и  N>0. }
   { Улучшенный алгоритм Эвклида. }
   var A1, A2, B1, B2, C, D, Q, R: Integer;
begin
   A1 := 0;  A2 := 1;  B1 := 1;  B2 := 0;  C := M;  D := N;
   while D <> 0 do
      begin
         {A1*M + B1*N = D, A2*M + B2*N = C, and GCD(C,D) = GCD(M,N)}
         Q := C div R;  R := C mod D;
         A2 := A2 - Q*A1;  B2 := B2 - Q*B1;
         C := D;  D := R;
         R := A1;  A1 := A2;  A2 := R;
         R := B1;  B1 := B2;  B2 := R
      end;
   X := C;  Y := A2;  Z := B2
   { X = GCD(M,N) = Y*M + Z*N }
end { GCD };
```

11.2. ОПИСАНИЯ ФУНКЦИЙ

Описание функции предназначено для введения имени функции, сопоставления с этим именем типа результата, блока и, возможно, списка формальных параметров. Заголовок функции из описания функции вводит имя функции, тип результата и список формальных параметров.

Функция может быть описана с помощью одного-единственного описания функции, состоящего из заголовка функции и блока. Это наиболее распространенный способ.

Однако есть и другой способ — функцию можно описывать и с помощью «опережающего описания»: первое описание состоит из заголовка функции и директивы forward, а второе описание

в том же самом разделе описания процедур и функций содержит наименование функции и блок. Имя функции в идентификации функции должно быть тем же, что и введенное в первом описании. Заметим, что список формальных параметров, если он есть, и тип результата во втором описании не указываются.

Описание функции = *Заголовок функции* ";" *Блок* |
Заголовок функции ";" *Директива* |
Идентификация функции ";" *Блок*.
Заголовок функции =
"function" *Имя* [*Список формальных параметров*] ":" *Тип результата*.
Тип результата = *Имя ординального типа* | *Имя вещественного типа* |
Имя ссылочного типа.
Идентификация функции = "function" *Имя функции*.
Имя функции = *Имя*.

Блок любого описания функции должен содержать по крайней мере один оператор присваивания, присваивающий (значение) имени функции. Использование имени функции в обозначении функции внутри блока ее описания предполагает рекурсивное выполнение этой функции.

Пример раздела описания процедур и функций, содержащего функцию:

```
function sqrt(X: Real): Real;
   { Метод Ньютона }
   var X0, X1: Real;
begin
   X1 := X;  { X > 1, метод Ньютона }
   repeat  X0 := X1;  X1 := (X0 + X/X0)*0.5
   until abs(X1 - X0) < Eps * X1;
   sqrt := X0
end { sqrt } ;
function Max(A: Vector;  N: Integer): Real;
   {  Максимальное значение из  A[1], ..., A[N]. }
   var X: Real;  I: Integer;
begin
   X := A[1];
   for I := 2 to N do
      begin  { X = Max( A[1], ..., A[I-1] ) }
         if X < A[I] then X := A[I]
      end;
```

```
    { X = Max( A[1], ..., A[N] ) }
    Max := X
end { Max } ;

function GCD(M, N: Natural): Natural;
begin if N = 0 then GCD := M else GCD := GCD(N, M mod N) end;

function Power(X: Real;  Y: Natural): Real;
   var W, Z: Real;  I: Natural;
begin
  W := X;  Z := 1;  I := Y;
   while I > 0 do
      begin
         { Z * (W ** I) = X ** Y }
         if odd(I) then Z := Z * W;
         I := I div 2;
         W := sqr(W)
      end;
   { Z = X ** Y }
   Power := Z
end { Power } ;
```

11.3. ПАРАМЕТРЫ

Параметры позволяют при каждой активации процедуры или функции работать с объектами (значениями, переменными, процедурами и функциями), задаваемыми в точке активации (см. разд. 10.3), через список фактических параметров. Список формальных параметров в заголовке процедуры или функции определяет имена, под которыми эти объекты известны в блоке процедуры или функции, а также природу и тип требуемых фактических параметров.

Фактические параметры для предописанных процедур и функций не всегда следуют (conform) правилам для обычных процедур и функций (см. разд. 11.4; 11.5 и 12).

11.3.1. Списки формальных параметров

Список формальных параметров = "("*Секция формальных параметров* {";" *Секция формальных параметров*}")".
Секция формальных параметров = *Спецификация параметров-значений* |
Спецификация параметров-переменных |
Спецификация процедурального параметра |
Спецификация функционального параметра.

Параметры, перечисленные в одной секции формальных параметров, представляют собой либо параметры-значения, либо пара-

метры-переменные, либо параметры-процедуры, либо параметры-функции.

11.3.1.1. Формальные параметры-значения и параметры-переменные. Спецификация параметров-значений и параметров-переменных в своем списке имен задает имена, рассматриваемые как имена переменных. Если указывается имя типа, то это значит, что к этому типу относится каждое из имен этих переменных. Если встречается схема совмещаемого массива, то каждое из имен переменных называется совмещаемым массивом-параметром, причем его тип зависит от типа фактического параметра. Внутри данной активации все формальные параметры, определенные в одной секции формальных параметров, относятся к одному типу.

Замечание. Не обязательно все реализации Паскаля будут поддерживать схемы совмещаемых массивов. В частности, реализации уровня 0 их не поддерживают, а уровня 1 — поддерживают.

Спецификация параметров-значений =
    Список имен ":" (*Имя типа*|*Схема совмещаемого массива*).
Спецификация параметра-переменной =
    "var" *Список имен* ":" (*Имя типа* |*Схема совмещаемого массива*).
Схема совмещаемого массива = *Схема упакованного совмещаемого массива*|
    Схема неупакованного совмещаемого массива.
Схема упакованного совмещаемого массива =
"packed" "array" "["*Спецификация типа индекса*"]" "of" *Имя типа.*
Схема неупакованного совмещаемого массива =
    "array" "["*Спецификация типа индекса* { ";" *Спецификация типа индекса*}"]" "of" (*Имя типа*|*Схема совмещаемого массива*).
Спецификация типа индекса = *Имя* ".." *Имя** ":" *Имя ординального типа.*
Имя границы = *Имя.*

Спецификация типа индекса вводит в качестве имен границ два имени, относящиеся к типу, обозначаемому именем ординального типа. Схема совмещаемого массива

array [Low1 .. High1 : T1; Low2 .. High2 : T2] of T

представляет собой просто сокращенную запись для

array [Low1 .. High1 : T1] of array [Low2 .. High2 : T2] of T

Пример описания функции, иллюстрирующий совмещаемый массив-параметр:

```
{  Этот пример получен из функции Max в 11.2 }

function Max (A: array [L..H: Integer] of Real;  N: Integer): Real;
  { Максимальное значение из  A[L],...,A[N]. }
  var X: Real;  I: Integer;
```

* Это, очевидно, неточность. Здесь должно стоять *Имя границы.* — *Примеч. пер.*

```
begin
   X:= A[L];
   for I := succ(L) to N do
      begin  { X = Max( A[L],...,A[I-1] ) }
         if X < A[I] then X := A[I]
      end;
   { X = Max( A[L],...,A[N] ) }
   Max := X
end { Max } ;
```

11.3.1.2. Формальные процедуральные параметры и функциональные параметры. Спецификация процедурального параметра вводит имя процедуры с каким-либо связанным с ним списком формальных параметров, определяемых заголовком процедуры.

Спецификация процедурального параметра = Заголовок процедуры.

Спецификация параметра-функции вводит имя функции с типом результата и каким-либо связанным с именем списком формальных параметров, определяемых заголовком функции.

Спецификация функционального параметра = Заголовок функции.

11.3.2. Списки фактических параметров. Список фактических параметров в точке активации, т. е. в операторе процедуры или обозначении функции, определяет фактические параметры, которые при этой активации должны быть подставлены в процедуре или функции вместо формальных параметров. Если у процедуры или функции нет списка формальных параметров, то не должно быть и списка фактических параметров. Соответствие между фактическими параметрами и формальными устанавливается путем позиционного сопоставления параметров из соответствующих списков. Порядок подстановки фактических параметров из списка зависит от реализации.

Список фактических параметров = "("*Фактический параметр* { "," *Фактический параметр*}")".
Фактический параметр = *Выражение*|*Переменная*|*Имя процедуры*|*Имя функции.*

Все фактические параметры из данной точки активации, соответствующие формальным совмещаемым массивам-параметрам которые определены в некоторой секции формальных параметров, должны относиться к типу, допускающему совмещение (см. разд. 11.3.4) со схемой совмещаемых массивов из упомянутой секции формальных параметров. В данной активации все соответствую-

щие формальные параметры относятся к некоторому типу, который выводится с помощью схемы совмещения из типа фактического параметра(ов) (см. разд. 11.3.4).

11.3.2.1. Фактические параметры-значения. Фактический параметр-значение — это некоторое выражение. Формальный параметр же обозначает некоторую переменную, которой при ее создании (см. разд. 10.3) присваивается значение фактического параметра.

Если формальный параметр не является совмещаемым массивом-параметром, то значение фактического параметра должно быть совместимо по присваиванию (см. разд. 6.5) с типом формального параметра.

Если формальный параметр — совмещаемый массив-параметр, то тип фактического параметра не должен быть совмещаемым типом (см. разд. 11.3.4).

11.3.2.2. Фактические параметры-переменные. Любой фактический параметр-переменная представляет собой переменную. На протяжении всей активации формальный параметр обозначает ту переменную, которую обозначал фактический параметр в момент начала активации (см. разд. 10.3). Фактический параметр не должен быть ни компонентой упакованного массива или записной переменной, ни полем признака.

Если формальный параметр не представляет собой совмещаемого массива-параметра, то и фактический параметр, и формальный параметр должны относиться к одному и тому же типу.

11.3.2.3. Фактические параметры-процедуры. Фактический параметр-процедура — это имя процедуры. Формальный параметр обозначает ту процедуру, которую обозначает такой фактический параметр (см. разд. 10.3). Списки формальных параметров, если они есть у формального и фактического параметров, должны быть конгруэнтными (см. разд. 11.3.3).

11.3.2.4. Фактические параметры-функции. Фактический параметр-функция представляет собой имя функции. Формальный параметр обозначает ту функцию, которую обозначает такой фактический параметр (см. разд. 10.3). Типы результата и формального, и фактического параметров должны обозначать один и тот же тип. Списки формальных параметров, если они есть у формального и фактического параметров, должны быть конгруэнтными (см. разд. 11.3.3).

11.3.3. Конгруэнтность списков параметров. Два списка формальных параметров называются *конгруэнтными,* если у них одинаковое число секций параметров, и сами соответствующие секции формальных параметров удовлетворяют одному из следующих условий:

а) обе представляют собой спецификации параметров-значений с одним и тем же числом имен в их списках имен, причем либо обе содержат имя типа, обозначающее один и тот же тип, либо обе содержат эквивалентные схемы совмещаемых массивов;

б) обе представляют собой спецификации параметров-переменных с одним и тем же числом имен в их списках имен, причем либо обе содержат имя типа, обозначающее один и тот же тип, либо обе содержат эквивалентные схемы совмещаемых массивов;

в) обе представляют собой спецификации параметров-процедур с конгруэнтными списками формальных параметров;

г) обе представляют собой спецификации параметров-функций с конгруэнтными списками формальных параметров и с типами результата, обозначающими один и тот же тип.

Две схемы совмещаемых массивов (каждая с одной единственной спецификацией типа индекса) называются *эквивалентными*, если справедливы все три следующих условия:

а) имена ординального типа в спецификации типа индекса обозначают один и тот же тип;

б) либо каждая содержит компоненту-схему схемы совмещаемых массивов, причем эти компоненты-схемы, эквивалентны, либо каждая содержит компоненту — имя типа, причем эти компоненты являются именами типов и обозначают один и тот же тип;

в) обе схемы представляют собой схемы либо упакованных, либо неупакованных совмещаемых массивов.

Пример двух эквивалентных схем совмещаемых массивов:

```
array [L1..H1: Integer; L2..H2: Color] of
                                    packed array [L3..H3: T2] of T

array [Low1..High1: Integer] of array [Low2..High2: Color] of
                                  packed array [Low3..High3: T2] of T
```

11.3.4. Совмещаемость и совмещаемые типы. Массивовый тип T (с единственным типом индекса) называется совмещаемым (conformable) со схемой S совмещаемого массива (с единственной спецификацией типа индекса), если справедливы все следующие условия. Пусть I — имя ординального типа из спецификации типа индекса в S.

а) Тип индекса из T совместим (compatible) с типом, обозначенным I.

б) Каждое из значений типа индекса из T представляет собой некоторый элемент множества значений, относящихся к типу, обозначенному I.

в) Если S не содержит какой-либо схемы совмещаемого массива, то тип компоненты из T тот же, что и тип, обозначенный именем типа в S; в противном случае тип компоненты из T — совмещаемый со схемой компоненты из S.

г) Тип T упакован, если и только если S представляет собой схему упакованного совмещаемого массива.

Ситуация, когда требуется совмещаемость, а условие б) не выполняется, считается ошибкой.

Совмещаемый тип, выведенный (derived) *с помощью S из T,* представляет собой массивовый тип, имеющий тот же тип индекса, что и T, причем он упакован, если и только если T — упакованный; тип его компонент или тот же самый, что и тип компонент из T, или же, если S содержит в качестве компоненты другую схему совмещаемого массива, представляет собой совмещаемый тип, выведенный с помощью схемы компоненты из типа компоненты из T. Имена границ, введенные в спецификации типа индекса, обозначают самое маленькое и самое большое значения, относящиеся к типу индекса из совмещаемого типа.

11.4. ПРЕДОПИСАННЫЕ ПРОЦЕДУРЫ

11.4.1. Процедуры для работы с файлами. Существует несколько предописанных процедур, введенных специально для работы с текстовыми файлами. Детально они описываются в разд. 12. Перечисленные ниже процедуры работают с любой файловой переменной f (см. разд. 6.2.4 и 7.4).

Rewrite(f)	приводит к тому, что f становится пустой последовательностью и находится в режиме формирования
Put(f)	ситуации, когда f не определена, не находится в режиме формирования или не определена буферная переменная f↑, считаются ошибкой. Процедура добавляет значение f↑ в конец последовательности f
Reset(f)	приводит к тому, что f переводится в режим чтения (inspection) и первая позиция последовательности становится ее текущей позицией. Если последовательность пуста, то eof(f) становится true, а f↑ — полностью неопределенной, в противном случае eof(f) становится false, а f↑ — первой компоненты последовательности
Get(f)	ситуация, когда f не определена или eof(f) — true, считается ошибкой. Обращение приводит к тому, что текущая позиция перемещается к следующей компоненте, если она существует, а f↑ получает ее значение; если очередной компоненты нет, то eof(f) становится true, а f↑ — полностью неопределенной

В каждом из следующих определений все вхождения f обозначают одну и ту же файловую переменную, имена v, v1, ..., vn соот-

ветствуют переменным, а e, e1, ..., en — выражениям. Заметим, что переменные v, v1, ..., vn не есть фактические параметры-переменные, поэтому они могут быть компонентами упакованных массивов и записей.

Read(f, v1, ..., vn)	эквивалентно оператору: begin Read(f, v1); ...; Read(f, vn) end
Read(f, v)	если f не текстовой файл, то обращение эквивалентно оператору: begin v := f↑; Get(f) end
Write(f, e1, ..., en)	эквивалентно оператору: begin Write(f, e1); ...; Write(f, en) end
Write(f, e)	если f не текстовой файл, то обращение эквивалентно оператору: begin f↑ := e; Put(f) end

11.4.2. Процедуры динамического размещения. Процедуры динамического размещения представляют собой средство, с помощью которого порождаются (New) новые ссылочные значения и соответствующие идентифицированные переменные или уничтожаются (Dispose). Пусть в последующих описаниях p — ссылочная переменная, q — ссылочное выражение, а c1, ..., cn, k1, ..., kn — константы. Заметим, что p не есть фактический параметр-переменная, поэтому оно может быть компонентой упакованного массива или записи.

New(p)	порождает новое идентифицирующее ссылочное значение, имеющее тип, указанный для p, и присваивает его переменной p. Идентифицированная переменная p↑ полностью не определена.
New(p, c1, ..., cn)	порождает новое идентифицированное ссылочное значение, имеющее тип, указанный для p, и присваивает его переменной p. Идентифицированная переменная p↑ полностью не определена. Типом области для этого ссылочного типа должен быть записной тип с вариантной частью. Первая из констант (c1) выбирает из этой вариантной части некоторый вариант; следующая константа (если она есть) выбирает вариант из следующей (вложенной) вариантной части и т. д. Если в идентифицированной переменной в этих вариантных частях будут сделаны активными какие-либо другие варианты, кроме выбранных, то это считается ошибкой. Ошибкой считается и использование идентифицированной переменной p↑ в качестве фактора, фактического параметра-переменной и в качестве переменной в операторе присваивания (однако компоненты p↑ в этих местах могут встречаться)
Dispose(q)	уничтожает идентифицирующее значение q. Если q равно nil, то такая ситуация считается ошибкой.

	Значение q должно быть порождено с помощью первого (короткого) варианта обращения к New, если это не так, то — ошибка
Dispose(q, k1, ..., kn)	уничтожает идентифицирующее значение q. Если q равно nil, то такая ситуация считается ошибкой. Значение q должно быть порождено с помощью второго (длинного) варианта обращения к New, причем k1, ..., kn должны выбирать те же варианты, которые были выбраны при порождении указанного значения, если это не так, то — ошибка

11.4.3. Процедуры передачи данных. Предположим, что U обозначает неупакованный массив (переменную), у которого тип индекса — S1, а тип компонент — T. Пусть P обозначает упакованный массив (переменную), у которого S2 — тип индекса, а T — тип компонент. B и C обозначают наименьшее и наибольшее значения, относящиеся к типу S2, K — некоторую новую переменную (нигде не используемую), относящуюся к типу S1, а J обозначает новую переменную, относящуюся к типу S2. Пусть I — некоторое выражение, совместимое с S1. Pack(U, I, P) эквивалентно оператору:

```
begin
   K := I;
   for J := B to C do
      begin
         P[J] := U[K];
         if J <> C then K := succ(K)
      end
end
```

Unpack (P, U, I) эквивалентно оператору

```
begin
   K := I;
   for J := B to C do
      begin
         U[K] := P[J];
         if J <> C then K := succ(K)
      end
end
```

И в том и в другом случае во всех итерациях оператора цикла P обозначает одну переменную и U — одну переменную.

11.5. ПРЕДОПИСАННЫЕ ФУНКЦИИ

11.5.1. Арифметические функции. Пусть x — любое вещественное или целое выражение. Тип результата функций abs и sqr тот же, что и тип x. Тип результата других арифметических функций — Real.

abs (x)	дает абсолютное значение x
sqr(x)	дает квадрат x. Ситуация, когда в данной реализации квадрата не существует, считается ошибкой
sin(x)	дает синус от x, где x выражено в радианах
cos(x)	дает косинус от x, где x — в радианах
exp(x)	дает значение, равное основанию натурального логарифма, возведенному в степень x
ln(x)	дает натуральный логарифм от x. Ситуация, когда x меньше или равно нулю, считается ошибкой
sqrt(x)	дает корень квадратный из x. Ситуация, когда x — отрицательное, считается ошибкой
arctan(x)	дает выраженное в радианах главное значение арктангенса от x.

11.5.2. Логические функции. Пусть x — любое целое выражение, а f обозначает любую переменную-файл. Тип результата всякой логической функции — Boolean.

odd(i)	эквивалентно выражению (abs(i) mod 2 = 1)
eof(f)	если f не определено, то — ошибка. Обращение eof(f) дает значение true, если f находится в режиме формирования или же до обращения f был установлен в последнюю позицию, т. е. на последнюю компоненту последовательности. Если список параметров (параметр) опущен, то eof применяется к параметру программы под именем Input
eoln(f)	если f не определено или eof(f) — истина, то — ошибка. Файл f должен быть текстовым файлом. Обращение eoln (f) дает значение true, если текущая компонента последовательности f представляет собой маркер конца строки. Если список параметров (параметр) опущен, то eoln применяется к параметру программы под именем Input

11.5.3. Преобразующие функции (transfer). Пусть r — любое вещественное выражение. Тип результата каждой из функций — Integer.

trunc(r)	дает значение, удовлетворяющее условию: $0 <= r - trunc(r) < 1$ при $r >= 0$ или же $-1 < r - trunc(r) <= 0$ при $r < 0$. Если такого значения не существует, то — ошибка
round(r)	дает значение, удовлетворяющее условию: round (r) = trunc (r + 0,5) при $r >= 0$, или же round (r) = trunc (r — 0,5) при $r < 0$. Если такого значения не существует, то — ошибка.

11.5.4. Ординальные функции. Пусть i — любое целое выражение, а x — любое ординальное выражение.

ord(x)	дает порядковый номер для x
chr(i)	дает значение типа Char с порядковым номером i. Если такого значения не существует, то — ошибка. Если c обозначает некоторое символьное значение, то всегда верно соотношение: chr(ord(c)) = c
succ(x)	дает следующее за x значение (если оно существует). В этом случае верно соотношение ord(succ(x)) = ord(x) + 1. Если следующего значения не существует, то — ошибка
pred(x)	дает значение, предшествующее x (если оно существует). В этом случае верно соотношение ord(pred(x)) = ord(x) — 1. Если предшествующего значения не существует, то — ошибка

12. ТЕКСТОВЫЕ ФАЙЛЫ. ВВОД И ВЫВОД

Разумные ввод и вывод базируются на текстовых файлах (см. разд. 6.2.4), задаваемых программе на Паскале в качестве параметров (программы). В окружении, где работает программа, им могут соответствовать разные вводные и выводные устройства вроде клавишной панели, дисплея, магнитной ленты или печатающего устройства. Для упрощения работы с текстовыми файлами вводятся три предописанные процедуры (Readln, Writeln и Page) и расширяются возможности двух других предописанных процедур (Read и Write; см. разд. 11.4.1). Текстовые файлы, с которыми работают все эти процедуры, не обязательно представляют собой вводные и выводные устройства, они могут быть и просто локальными файлами. Списки фактических параметров этих процедур не следуют обычным правилам (см. разд. 11.3); кроме всего прочего, в них (списках) может быть разное (переменное) число параметров. Кроме того, параметры не обязательно относятся к типу Char. Они могут быть и других типов. В этом случае при передаче данных выполняется неявная операция преобразования, Если первый параметр — файловая переменная, то ввод или вывод идет именно в этот файл. В противном же случае считается, что ввод идет из файла (параметра программы) Input, а вывод — в файл (параметр программы) Output (см. разд. 13).

12.1. ЧТЕНИЕ (READ)

Работа процедуры Read с текстовым файлом следует таким правилам. Пусть f обозначает текстовый файл, а v1, ..., vn —

переменные, относящиеся к типам Char, Integer (либо их диапазонам) или Real.

а) Read (v1, ..., vn) эквивалентно Read (g, v1, ..., vn), где g обозначает параметр программы — файл Input.

б) Read (f, v1, ..., vn) эквивалентно оператору:

begin Read (f, v1); ...; Read (f, vn) end

где все вхождения f обозначают одну и ту же переменную.

в) Обращение Read (f, v) считается ошибкой, если f не определено либо же находится в режиме просмотра, или же eof (f) — true. Действие обращения Read (f, v) зависит от типа параметра v.

12.1.1. Чтение символа. Обращение Read (f, v), где v обозначает переменную, относящуюся к типу, совместимому с типом Char, эквивалентно оператору:

begin v := f↑; Get(f) end

где все вхождения f обозначают одну и ту же переменную. Если перед обращением Read(f, v) было истинно eoln(f), то после будет истинно условие (v = ' ').

12.1.2. Чтение целого числа. Обращение Read(f, v), где v обозначает переменную, относящуюся к типу, совместному с типом Integer, предполагает, что из f читается последовательность символов, образующая *целое число со знаком* (см. разд. 4), целое значение которого присваивается переменной v. Предшествующие числу пробелы и маркеры конца строки пропускаются. Если целое число со знаком не обнаружено, то — ошибка.

12.1.3. Чтение вещественного числа. Обращение Read(f, v), где v обозначает переменную, относящуюся к типу Real, предполагает, что из f читается последовательность символов, образующая *число со знаком* (см. разд. 4), вещественное значение которого присваивается переменной v. Предшествующие числу пробелы и маркеры конца строки пропускаются. Если число со знаком не обнаружено, то — ошибка.

12.2. ЧТЕНИЕ СТРОКИ (READLN)

Пусть f обозначает некоторый текстовый файл, а v1, ..., vn — переменные типа Char или Integer (либо их диапазоны) или Real.

Обращение Readln(v1, ..., vn) эквивалентно Readln(g, v1, ..., vn), а Readln эквивалентно Readln(g), где g обозначает параметр программы — текстовый файл Input.

Обращение Readln(f, v1, ..., vn) эквивалентно оператору:

begin Read(f, v1, ..., vn); Readln(f) end

где все вхождения f обозначают одну и ту же переменную.

Обращение Readln(f) эквивалентно оператору:

```
begin
   while not eoln(f) do Get(f);
   Get(f)
end
```

где все вхождения f обозначают одну и ту же переменную.

12.3. ЗАПИСЬ (WRITE)

Работа процедуры Write с текстовым файлом следует таким правилам. Пусть f обозначает некоторый текстовый файл, p, p1, ..., pn — *параметры вывода*, e — выражение, а m и n — целые выражения. Список фактических параметров вывода должен удовлетворять такому синтаксису.

Список параметров вывода = "(" (*Переменная-файл*| *Параметр вывода*) {"," *Параметр вывода*}")".
Параметр вывода = *Выражение* [":" *Целое выражение* [":" *Целое выражение*]].

а) Обращение Write(p1, ..., pn) эквивалентно Write(g, p1, ..., pn), где g — параметр программы (текстовый файл Output).

б) Обращение Write(f, p1, ..., pn) эквивалентно оператору:

begin Write(f, p1); ...; Write(f, pn) end

где все вхождения f обозначают одну и ту же переменную.

в) Если f не определено или не находится в режиме формирования, то обращение Write(f, p) — ошибка.

г) Любой параметр записи имеет один из таких видов:

e e:m e:m:n

причем e представляет собой значение, «записываемое» в f, а m и n — так называемые параметры, размеры поля (field-width parameters). Если n или m меньше или равно нулю, то это ошибка. Выражение e должно относиться к типу Integer, Real, Char, Boolean или же к некоторому строковому типу. Выражение n может встречаться только в том случае, если e — вещественного (Real) типа (см. разд. 12.3.3.). Если m опущено, то вместо него предполагается некоторое подразумеваемое значение. Подразумеваемое значение, если речь идет о типах Integer, Real или Boolean, определяется при реализации. Для типа Char подразумеваемым значением является 1, а для строкового типа оно равно числу элементов в строке.

Если для представления значения e требуется меньше, чем m символов, то ему предшествует такое число пробелов, чтобы было записано точно m символов. Представление значения e зависит от типа e.

12.3.1. Запись символа. Если e относится к типу Char, то обращение Write(f, e : m) эквивалентно оператору:

```
begin
   for J := 1 to m - 1 do Write(f,' ');
   ff := e; Put(f)
end
```

где все вхождения f обозначают одну и ту же переменную, а J обозначает новую (нигде ранее не встречающуюся) целую переменную.

12.3.2. Запись целого числа. Если e относится к типу Integer, то при e < 0 обращение Write(f, e : m) записывает «—» (минус), за которым следует десятичное представление abs(e). Если необходимо, то записываются предшествующие пробелы (нужно чтобы было всего m символов).

12.3.3. Запись вещественного числа. Если e относится к типу Real, то обращение Write(f, e : m : n) записывает с n цифрами после десятичной точки представление с фиксированной точкой, а Write(f, e : m) записывает представление с плавающей точкой соответствующего значения. Операция «**» означает «возведение в степень».

12.3.3.1. Представление с фиксированной точкой. Пусть, если e — нуль, w будет нулевым, в других же случаях w — абсолютное значение, округленное, а затем сокращенное до n десятичных позиций. Пусть если w < 1, d будет 1, в других же случаях пусть d (удовлетворяет условию. — *Примеч. пер.*) 10**(d — 1) <= w < 10**d. Значение d — число цифр влево от десятичной точки. Пусть s = ord((e < 0) and (w <>0)). Если s = 1, то представление — отрицательное. Пусть k = (s + d + 1 + n), где k — число записываемых, отличных от пробела символов. Если k < m, то записывается m — k предшествующих пробелов. Представление e с фиксированной точкой состоит из k символов:

а) '—', если s = 1;

б) d десятичных цифр целой части w;

в) '.';

г) n старших десятичных цифр дробной части w.

12.3.3.2. Представление с плавающей точкой. Число цифр, помещаемых в порядок представления с плавающей точкой, определяется при реализации; это число будем обозначать через x. Пусть k — наибольшее между m и x + 6. Число значащих цифр, которое нужно записать: k — x — 4, т. е, записывается одна цифра перед десятичной точкой и d цифр после (таким образом, d = k — x — — 5). Предположим, если e равно нулю, w и s будут нулевыми. Если же e отлично от нуля, то s будет удовлетворять условию

10.0** s <=abs(e) < 10.0** (s + 1), а w равно (abs(e)/ 10.0** s) + 0.5*10.0*(—d). Если w >= 10.0, то w и s должны быть изменены на s := s + 1 и w := w/10.0. И наконец, w сокращается до d десятичных позиций.

Представление e с плавающей точкой состоит из:

а) '—', если ((e < 0) and (w <>0)), иначе ' ';

б) старшей значащей десятичной цифры w;

в) '.';

г) d последующих значащих десятичных цифр w;

д) либо 'e', либо 'E' (это определяется при реализации);

е) '—', если s < 0; иначе '+';

ж) x десятичных цифр с s (если нужно) начальными нулями.

12.3.4. Запись логического значения. Если e относится к типу Boolean, то оператор Write(f, e : m) записывает представление слов true или false. Этот оператор эквивалентен оператору:

if then Write(f, 'true' : m) else Write(f, 'false' : m)

однако регистр, на котором записываются буквы, определяется при реализации.

12.3.5. Запись строки. Если e относится к строковому типу с длиной строки k, то при Write(f, e : m) записывается m — k пробелов, если m > k, а затем идут компоненты (строки) с последовательными индексами, начиная с 1 и до меньшего из k и m.

12.4. ЗАПИСЬ СТРОКИ ТЕКСТА (WRITELN)

Пусть f — текстовый файл, а p1, ..., pn — параметры вывода.

Оператор Writeln(p1, ..., pn) эквивалентен оператору Writeln (g, p1, ..., pn), а Writeln эквивалентен Writeln(g), где g обозначает текстовый файл — параметр программы под именем Output. Обращение Writeln(f, p1; ..., pn) эквивалентно оператору:

begin Write(f, p1, ..., pn); Writeln(f) end

где все вхождения f обозначают одну и ту же переменную.

Обращение Writeln(f) добавляет маркер конца строки к последовательности (символов) файла f. Если f не определено или находится в режиме формирования, то — ошибка.

12.5. СТРАНИЦА (PAGE)

Оператор Page(f) вызывает для текстового файла f некоторый определяемый при реализации эффект, заключающийся в том, что любой текст, затем записываемый в f, при печати f будет появляться, начиная с новой страницы. Если f не пуст и последняя его компонента не маркер конца строки, то Page(f) выполняет неявно

Writeln(f). Если список параметров* опущен, то предполагается, что речь идет о текстовом файле — параметре программы под именем Output. Если f не определено или не находится в режиме формирования, то — ошибка.

Эффект чтения файловой переменной, к которой ранее применялась процедура Page, определяется при реализации.

13. ПРОГРАММЫ

Программа на языке Паскаль состоит из заголовка программы и блока.

Программа = *Заголовок программы* ";" *Блок* ".".
Заголовок программы = "program" *Имя* [*Список параметров программы*].
Список параметров программы = "("*Список имен*")".

Имя, идущее следом за словом program, и есть имя программы; внутри программы оно не имеет никакого смысла. Каждое из имен в списке параметров программы называется параметром программы и обозначает объект, существующий вне этой программы, поэтому такие объекты называются *внешними*. Через параметры программы сами связываются с внешним окружением.

При активации программы каждый из параметров программы связывается с внешним объектом, который он представляет. Для параметров программы, представляющих собой файловые переменные, процесс связывания определяется при реализации, для всех же других параметров программы этот процесс зависит от реализации.

Каждый параметр программы, за исключением Input и Output, должен быть описан в разделе описания переменных блока самой программы. Что же касается Input и Output, то появление такого имени в списке параметров программы неявно описывает в блоке программы это имя как текстовый файл, и в начале каждой активации программы неявно выполняется Reset(Input) или Rewrite (Output).

Эффект применения Reset или Rewrite к Input или Output определяется при реализации.

---

* Хотя в некоторых других местах мы при переводе заменяли «список параметров» на «параметр», здесь мы его оставим, чтобы продемонстрировать излишнюю «синтаксическую ориентированность» нового описания. — *Примеч. пер.*

Примеры программ:

```
program CopyReals(F,G);
   var F, G: file of Real; R: Real;
begin
   Reset(F); Rewrite(G);
   while not eof(F) do
      begin  Read(F,R); Write(G,R)  end
end { CopyReals }

program CopyText(Input,Output);
begin
   while not eof(Input) do
      begin
         while not eoln(Input) do
            begin Input↑ := Output↑; Put(Output); Get(Input) end;
         Readln(Input); Writeln(Output)
      end
end { CopyText }
```

14. СОГЛАСОВАННОСТЬ СО СТАНДАРТОМ ИСО 7185

Программа удовлетворяет стандарту ИСО Паскаля [11], если в ней используются те свойства языка, которые определены в стандарте и она не ориентирована (rely) на какие-либо частные интерпретации особенностей, зависящих от реализации. Будем говорить, что программа удовлетворяет уровню 1, если она такие особенности не использует.

Как определяется в стандарте, *процессор* — это «некоторая система или механизм, который в качестве входа воспринимает программу, подготавливает ее для выполнения, и выполняет процесс, зависящий от данных, который и порождает результаты». Процессор согласован со стандартом, если он удовлетворяет следующим условиям.

1. Воспринимает все особенности языка такими, какими они определены в стандарте. Будем говорить, что процессор соответствует уровню 0, если он не воспринимает совмещаемые массивы-параметры, и уровню 1, если он их воспринимает.

2. Не требует использования заменяющих (substitute) или дополнительных элементов языка для того, чтобы можно было воспользоваться особенностями самого языка.

3. Способен определять отклонения от стандарта, которые специально не выделены как ошибки, и сообщать о них пользователю. Если процессор не проверяет все программы на такие отклонения, то он должен сообщать об этом факте.

4. На каждое из отклонений, специально выделенных как ошибки, он должен реагировать одним из следующих способов:

а) в его документации должно быть сказано, что об ошибке сообщение не выдается;

б) в процессе подготовки программы сообщается, что возможно есть ошибка;

в) в процессе подготовки программы сообщается, что ошибка «будет»;

г) в процессе выполнения программы сообщается, что встретилась ошибка.

5. Способен обрабатывать как ошибку любое использование какого-либо расширения или свойства, зависящего от реализации.

6. Сопровождается документацией, содержащей:

а) определения всех свойств, фиксированных при реализации;

б) раздел, где описываются все ошибки, о которых сообщения не выдаются (см. выше п. 4а). Если некоторые расширения используют ситуацию, не определенную в стандарте как ошибка, т. е. сообщения о ней не выдаются, то в документации должно быть сказано, что сообщение об этой ошибке не дается;

в) раздел, где перечисляются все расширения, поддерживаемые данной реализацией.

Литература

1. N. Wirth. „The Programming Language Pascal", *Acta Informatica,* 1, 35—63, 1971.
2. N. Wirth. „Program Development by Stepwise Refinement", *Communications of the ACM 14,* 221—227, April 1971.
3. N. Wirth. *Systematic Programming,* Prentice-Hall, Inc. 1973. Русский перевод: Вирт Н. Систематическое программирование: Введение. — М.: Мир, 1977.
4. O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming,* Academic Press Inc. 1972.
   Русский перевод: Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975.
5. C. A. R. Hoare and N. Wirth. „An Axiomatic Definition of the Programming Language Pascal", *Acta Informatica,* 2, 335—355, 1973.
6. D. E. Knuth. The Art of Computer Programming, vol 1. *Fundamental Algorithms,* Addison-Wesley, 1968.
   Русский перевод: Кнут Д. Искусство программирования для ЭВМ: Основные алгоритмы. — М.: Мир, 1976.
7. N. Wirth. „An Assessment of the Programming Language Pascal", SIGPLAN Notices, *10,* 23—30, June 1975.
8. N. Wirth. „The Design of a Pascal Compiler", *SOFTWARE — Practice and Experience, 1,* 309—333, 1971.
9. N. Wirth. *Algorithms + Data Structures = Programs,* Prentice Hall, Inc., 1976.
   Русский перевод: Вирт Н. Алгоритмы + структура данных = программы. — М.: Мир, 1985.
10. D. Barron. „A Perspective on Pascal" and J. Welsh, W. Sneeringer, and C. A. R. Hoare. „Ambiguities and Insecurities in Pascal" *Pascal—The Language and its Implementation,* John Wiley, 1981.
11. International Organization for Standardization, *Specification for Computer Programming Language Pascal,* ISO 7185—1982, 1982.
12. A. H. J. Sale and B. Wichmann. „The Pascal Validation Suite", *Pascal News 16,* 5—153, 1979.
13. N. Wirth. „What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" *Communications of the ACM 20,* 822—823, November, 1977.
14. B. Wichmann and Z. J. Ciechanowicz. *Pascal Compiler Validation,* John Wiley, 1983.

ПРИЛОЖЕНИЕ 1

ПРЕДОПИСАННЫЕ ПРОЦЕДУРЫ И ФУНКЦИИ

Abs(x)

арифметическая функция, вычисляющая вещественное абсолютное значение вещественного параметра x или целое абсолютное значение целого параметра x.

ArcTan(x)

арифметическая функция, вычисляющая вещественное значение (в радианах) арктангенса (главное значение) для вещественного или целого параметра x.

Chr(i)

функция преобразования, дающая символ, порядковый номер которого равен целому параметру i. Если такого символа не существует, то Chr(i) — ошибка.

Dispose(q)

процедура динамического размещения, убирающая идентифицированную переменную q↑ и уничтожающая идентифицирующее значение q. Если q равно nil или не определено, то Dispose(q) — ошибка. Значение q должно быть порождено короткой формой обращения к New.

Dispose(q, k1, ..., kn)

процедура динамического размещения, убирающая идентифицированную записную переменную q↑ и уничтожающая идентифицирующее значение q. Если q равно nil или не определено, то Dispose(q, k1, ..., kn) — ошибка. Значение q должно быть порождено длинной формой обращения к New, причем k1, ..., kn должны выбирать те же самые варианты, которые были выбраны при порождении q.

Eof(f)

логическая функция, для файловой переменной f, дающая значение true, если f находится в режиме формирования или же в режиме просмотра, либо если файл стоит после последней компоненты последовательности. Если f не определено, то обращение eof(f) — ошибка. Во всех других случаях функция дает значение false. Если f опущено, то подразумевается параметр программы с именем Input.

Eoln(f)

логическая функция дающая значение true, если текстовый файл находится в режиме просмотра и стоит на маркере конца строки. Если f не определено или же eof(f) становится true, то eoln(f) — ошибка. В других случаях eoln(f) дает значение false. Если f опущено, то подразумевается параметр программы с именем Input.

Exp(x)

арифметическая функция, вычисляющая вещественное значение e (основание натурального логарифма), возведенное в степень, равную вещественному или целому параметру x.

Get(f)

процедура работы с файлами, приводящая к переходу на следующую компоненту последовательности, если она есть, и при этом f↑ принимает значение этой компоненты. Если следующей компоненты не существует, то eof(f) становится true, а f↑ — полностью неопределенным. Если f не определено или eof(f) становится true, то Get(f) — ошибка. Если f опущено, то подразумевается параметр программы с именем Input.

Ln(x)

арифметическая функция, вычисляющая вещественное значение натурального логарифма (с основанием e) вещественного или целого параметра x, при $x > 0$. Если $x <= 0$, то Ln(x) — ошибка.

New(p)

процедура динамического размещения новой идентифицированной (динамической) переменной p↑, относящейся к типу области из p, которая порождает новое идентифицирующее ссылочное значение, относящееся к типу p; это значение присваивается p. Если p↑ — вариантная запись, то New(p) выделяет пространство, достаточное для размещения всех вариантов.

New(p, c1, ..., cn)

процедура динамического размещения новой идентифицированной (динамической) переменной p↑, относящейся к вариантному записному типу из p со значениями полей признаков — c1, ..., cn для n вложенных вариантны частей, которая порождает новое идентифицирующее ссылочное значение, относящееся к типу p; это значение присваивается p.

Odd(i)

логическая функция, дающая значение true, если целый параметр не делится на 2, т. е. нечетен. В противном случае возвращается значение false.

Ord(x)

функция преобразования, дающая порядковый номер (целое значение) ординального параметра среди множества значений, определенных типов, к которому относится x.

Pack (u, i, p)

процедура передачи данных, упаковывающая, начиная с i-й компоненты, неупакованный массив u в упакованный массив p.

Page(f)

процедура работы с файлом, вызывающая определяемое при реализации действие, относящееся к текстовому файлу f, которое заключается в том, что записанный впоследствии в f любой текст при последующей печати будет появляться в начале новой страницы. Если f не пуст и последняя компонента его последовательности не является маркером конца строки, то Page(f) неявно выполняет Writeln(f). Если список параметров опущен, то подразумевается текстовый файл — параметр программы с именем Output. Если f не определено или не находится в режиме формирования, то обращение Page(f) — ошибка.

Pred(x)

ординальная функция, дающая ординальное значение, предшествующее ординальному параметру x; если такой «предшественник» существует, то ord(pred(x)) = ord(x) — 1. Если x — наименьшее значение соответствующего типа, то обращение Pred(x) — ошибка.

Put(f)

процедура работы с файлом, добавляющая значение f↑ в конец последовательности в f. Если f не определено или не находится в режиме формирования либо же не определена буферная переменная f↑, то обращение Put(f) — ошибка. После Put(f) f↑ полностью не определено.

Read(f, v)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 11.4 и 12.1.

Read(f, v1, ..., vn)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 11.4 и 12.1.

Readln

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 12.2.

Readln(f, v1, ..., vn)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 12.2.

Reset(f)

процедура работы с файлом, переводящая f в режим просмотра и ставящая его в первую позицию. Если f — пуст, то eof(f) становится true, а f↑ — полностью не определено. В противном случае eof(f) становится false, а f↑ принимает значение первой компоненты последовательности.

Rewrite(f)

процедура работы с файлом, заменяющая f на пустую последовательность и переводящая его в режим формирования. Eof(f) становится true.

Round(n)

функция преобразования, дающая при вещественном параметре r >=0.0 значение trunc(r + 0.5), а при r < 0.0 — значение trunc(r — 0.5), если, конечно, в типе Integer такие значения существуют. Если это не так, то — ошибка.

Sin(x)

арифметическая функция, вычисляющая вещественное значение синуса от вещественного или целого аргумента x, где x выражено в радианах.

Sqr(x)

арифметическая функция, вычисляющая вещественное значение x*x, если x — вещественное, и целое значение x*x, если x — целое. Если такого значения не существует, то — ошибка.

Sqrt(x)

арифметическая функция, вычисляющая вещественное, неотрицательное значение корня квадратного из целого или вещественного параметра x при x >= 0. Если x < 0, то обращение Sqrt(x) — ошибка.

Succ(x)

ординальная функция, дающая следующее ординальное значение, идущее после ординального параметра x; если такой «последователь» существует, то ord(succ(x)) = ord(x) + 1. Если x — максимальное значение соответствующего типа, то обращение succ(f) — ошибка.

Trunc(r)

функция преобразования, вычисляющая наибольшее целое число, меньшее или равное вещественному параметру r при r >= = 0.0. Если же r < 0.0, то она дает наименьшее целое число, большее или равное параметру r (если, конечно, такое значение для типа Integer существует). Если это не так — ошибка.

Unpack(*p*, *u*, *i*)

функция передачи данных, распаковывающая упакованный массив в неупакованный массив u, начиная с i-го элемента неупакованного массива.

Write(*f*, *v*)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 11.4 и 12.3.

Write(*f*, *v*1, ..., *vn*)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 11.4 и 12.3.

Writeln

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 12.4.

Writeln(*f*, *e*1; ..., *en*)

см. «Руководство для пользователя», гл. 9 и 12; «Описание языка», разд. 12.4.

ПРИЛОЖЕНИЕ 2

СВОДКА ОПЕРАЦИЙ

Операция	*Действие*	*Тип операнда*	*Тип результата*
Арифметические			
+ (унарный)	тождественное	Integer или Real	такой же, как и у операнда
— (унарный)	изменение знака	Integer или Real	такой же, как и у операнда
+	сложение	Integer или Real	Integer
—	вычитание		или Real
*	умножение		
div	целое деление	целый	Integer
/	вещественное деление	вещественный или целый	Real
mod	остаток	целый	Integer
Отношения			
=	равенство	простой, строковый	Boolean
<>	неравенство	множественный или ссылочный	Boolean
<	меньше	простой или строковый	Boolean
>	больше		
<=	меньше или равно	простой или строковый	Boolean
	либо включение	множественный	
>=	больше или равно	простой или строковый	Boolean
	либо включение	множественный	
in	присутствие в множестве	первый операнд любого ординального базового типа, второй — порожденного множественного типа	Boolean

Продолжение

Операция	*Действие*	*Тип операнда*	*Тип результата*
Логические			
not	отрицание	Boolean	Boolean
or	дизъюнкция	»	»
and	конъюнкция	»	»
Множественные			
+	объединение	любой множественный тип T	T
−	разность множеств		
*	пересечение		

Старшинство операций в выражениях

Операция	*Класс*
not	Логическое отрицание
* / div mod and	Операции умножения (мультипликативные)
+ − or	Операции сложения (аддитивные)
= <> > < >= <= in	Операции отношения

Другие операции

Операция	*Действие*	*Тип операнда*	*Тип результа*
Присваивание			
:=	присваивание	любой присваиваемый тип	нет
Обращение к переменным			
[,]	индексация массива	массивовый	тип компоненты
.	выбор поля	записной	тип поля
↑	идентификация	ссылочный	тип области
↑	обращение к буферу	файловый	тип компоненты
Конструктор			
[,]	конструктор множества	базовый тип	множественный
' '	конструктор строк	символьный	строковый

ПРИЛОЖЕНИЕ 3

ТАБЛИЦЫ

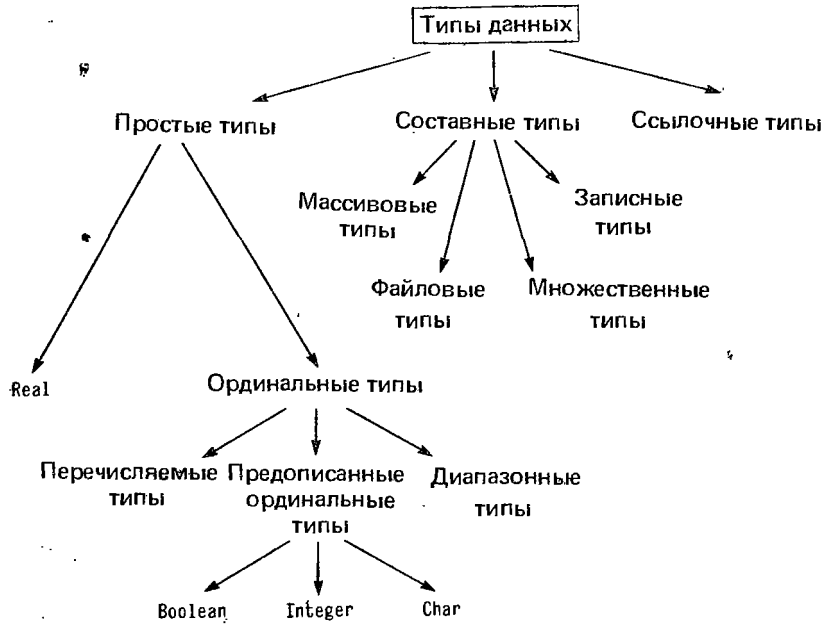


Рис. П.3.1. Полная схема типов данных

Таблица стандартных имен

Константы:

```
False, MaxInt, True
```

Типы:

```
Boolean, Char, Integer, Real, Text
```

Переменные:

```
Input, Output
```

Функции:

```
Abs, ArcTan, Chr, Cos, Eof, Eoln, Exp, Ln, Odd,
Ord, Pred, Round, Sin, Sqr, Sqrt, Succ, Trunc
```

Процедуры:

```
Dispose, Get, New, Pack, Page, Put, Read, Readln,
Reset, Rewrite, Unpack, Write, Writeln
```

Алфавитный список:

Abs	False	Pack	Sin
ArcTan	Get	Page	Sqr
Boolean	Input	Pred	Sqrt
Char	Integer	Put	Succ
Chr	Ln	Read	Text
Cos	MaxInt	Readln	True
Dispose	New	Real	Trunc
Eof	Odd	Reset	Unpack
Eoln	Ord	Rewrite	Write
Exp	Output	Round	Writeln

Таблица символов

Специальные символы:

```
+   -   *   /   =
<   >   <=  >=  <>
.   ,   :   ;   :=  ..
(   )   [   ]   ↑
```

Символы-слова (зарезервированные слова):

and	end	nil	set
array	file	not	then
begin	for	of	to

case	function	or	type
const	goto	packed	until
div	if	procedure	ver
do	in	program	while
downto	label	record	with
else	mod	repeat	

Альтернативное представление:

(.	for	[
.)	for	]
@	for	↑

Директивы:

forward

ПРИЛОЖЕНИЕ 4

СИНТАКСИС

Описание синтаксиса языка программирования с помощью Расширенных Бэкуса—Наура Форм (РБНФ) состоит из набора правил, иногда называемых и «продукциями». определяющими процесс образования предложений в языке. Множество таких правил называют «грамматикой». Каждое правило состоит из нетерминального символа и РБНФ-выражения, разделенных знаком равенства, в конце правила ставится точка. Нетерминальный символ представляет собой некоторое «метаимя» (синтаксическую константу, обозначаемую с помощью английского слова), а РБНФ-выражение — определение этого метаимени.

РБНФ-выражение включает нуль или более терминальных символов, нетерминальные символы и другие метасимволы, приводимые в нижележащей таблице.

Метасимвол	**Значение**
=	равно по определению
|	или (альтернатива)
.	конец правила
[X]	0 или 1 вхождение X
{X}	0 или более вхождений
(X|Y)	группирование: или X, или Y
"XYZ"	терминальный символ XYZ
Метаимя	нетерминальный символ с именем *Метаимя*

Правила РБНФ можно, например, использовать для определения их собственного синтаксиса.

Синтаксис = {*Правило*}.
Правило = *Нетерминал* "=" *Выражение* "."
Выражение = *Терм* { "|" *Терм*}.
Терм = *Фактор* {*Фактор*}.
Фактор = *Нетерминал*| *Терминал*| "("*Выражение*")"|
"["*Выражение*"]"|"{"*Выражение*"}".
Терминал = """ *Символ* {*Символ*} """
Нетерминал = *Буква* {*Буква*| *Цифра*}.

Замечания.

1. Любой терминальный символ (буквальное его изображение — литерал) всегда заключается в двойные кавычки. Если же заключаются и сами двойные кавычки, то их записывают дважды. Таким образом, в следующем далее описании Паскаля с помощью РБНФ "[" и "]" в программе на Паскале представляют левую и правую квадратные скобки, а [и] — метасимволы из РБНФ-выражений, указывающие на нуль или одно вхождение того, что в них заключено.

2. В каждом синтаксисе есть начальный символ (метаимя), из которого выводятся все предложения языка. В синтаксисе для Паскаля начальный символ — *Программа*.

СИНТАКСИС ЯЗЫКА, ЗАПИСАННЫЙ С ПОМОЩЬЮ ПРАВИЛ РБНФ

1. *Программа* = *Заголовок программы* ";" *Блок* ".".
2. *Заголовок программы* = "program" *Имя* [*Список параметров программы*].
3. *Список параметров программы* = "("*Список имен*")".
4. *Блок* = *Раздел описания меток*
   Раздел определения констант
   Раздел определения типов
   Раздел описания переменных
   Раздел описания процедур и функций
   Раздел операторов
5. *Раздел описания меток* = ["label" *Последовательность цифр* {"," *Последовательность цифр*}";"].
6. *Раздел определения констант* = ["const" *Определение константы* ";" {*Определение константы* ";"}].
7. *Раздел определения типов* = ["type" *Определение типа* ";" {*Определение типа* ";"}].
8. *Раздел описания переменных* = ["var" *Описание переменной* ";" {*Описание переменной* ";"}].
9. *Раздел описания процедур и функций* = {(*Описание процедуры* | *Описание функции*) ";"}.
10. *Раздел операторов* = *Составной оператор*.
11. *Определение константы* = *Имя* "=" *Константа*.
12. *Определение типа* = *Имя* "=" *Тип*.
13. *Описание переменной* = *Список имен* ":" *Тип*.
14. *Описание процедуры* = *Заголовок процедуры* ";" *Блок* |
    Заголовок процедуры ";" *Директива* |
    Идентификация процедуры ";" *Блок*.

15. *Описание функции* = *Заголовок функции* ";" *Блок* |
Заголовок функции ";" *Директива* |
Идентификация функции ";" *Блок*.
16. *Заголовок процедуры* = "procedur" *Имя* [*Список формальных параметров*].
17. *Идентификация процедуры* = "procedur" *Имя процедуры*.
18. *Заголовок функции* = "function" "*Имя* [*Список формальных параметров*] ";" *Тип результата*.
19. *Идентификатор функции* = "function" *Имя функции*.
20. *Список формальных параметров* = "("*Секция формальных параметров* {";" *Секция формальных параметров*} ")".
21. *Секция формальных параметров* = *Спецификация параметров-значений* | *Специализация параметров-переменных* | *Спецификация процедурального параметра* | *Спецификация функционального параметра*.
22. *Спецификация параметров-значений* = *Список имен* ":". (*Имя типа* | *Схема совмещаемого массива*).
23. *Спецификация параметров-переменных* = "var" *Список имен* ":" (*Имя типа* | *Схема совмещаемого массива*).
24. *Спецификация процедурального параметра* = *Заголовок процедуры*.
25. *Спецификация функционального параметра* = *Заголовок функции*.
26. *Схема совмещаемого массива* = *Схема упакованного совмещаемого массива* | *Схема неупакованного совмещаемого массива*.
27. *Схема упакованного совмещаемого массива* = "packed" "array" "["*Спецификация типа индекса*"]" "of" *Имя типа*.
28. *Схема неупакованного совмещаемого массива* = "array" "["*Спецификация типа индекса* { ";" *Спецификация типа индекса* } "]", "of" (*Имя типа* | *Схема совмещаемого массива*).
29. *Спецификация типа индекса* = *Имя* ".." *Имя* ":" *Имя ординального типа*.
30. *Составной оператор* = "begin" *Последовательность операторов* "end".
31. *Последовательность операторов* = *Оператор* {";" *Оператор*}.
32. *Оператор* = [*Метка* ":"] (*Простой оператор* | *Сложный оператор*).
33. *Простой оператор* = *Пустой оператор* | *Оператор присваивания* | *Оператор процедуры* | *Оператор перехода*.

34. *Сложный оператор* = *Составной оператор* | *Выбирающий оператор* | *Циклический оператор* | *Оператор присоединения*.
35. *Выбирающий оператор* = *Условный оператор* | *Оператор варианта*.
36. *Циклический оператор* = *Цикл с предусловием* | *Цикл с постусловием* | *Цикл с шагом*.
37. *Пустой оператор* =.
38. *Оператор присваивания* = (*Переменная* | *Имя функции*) "=" *Выражение*.
39. *Оператор процедуры* = *Имя процедуры* [*Список фактических параметров* | *Список параметров вывода*].
40. *Оператор перехода* = "goto" *Метка*.
41. *Условный оператор* = "if" *Логическое выражение* "then" *Оператор* ["else" *Оператор*].
42. *Оператор варианта* = "case" *Индекс варианта* "of" *Вариант* {";" *Вариант*} [";"] "end".
43. *Цикл с постусловием* = "repeat" *Последовательность операторов* "until" *Логическое выражение*.
44. *Цикл с предусловием* = "while" *Логическое выражение* "do" *Оператор*.
45. *Цикл с шагом* = "for" *Параметр цикла* ":=" *Начальное значение* ("to" | "downto") *Конечное значение* "do" *Оператор*.
46. *Оператор присоединения* = "with" *Список переменных-записей* "do" *Оператор*.
47. *Список переменных-записей* = *Переменная-запись* {"," *Переменная-запись*}.
48. *Индекс варианта* = *Ординальное выражение*.
49. *Вариант* = *Константа* {"," *Константа*} ":" *Оператор*.
50. *Параметр цикла* = *Имя переменной*.
51. *Начальное значение* = *Ординальное выражение*.
52. *Конечное значение* = *Ординальное выражение*.
53. *Тип* = *Простой тип* | *Составной тип* | *Ссылочный тип*.
54. *Простой тип* = *Ординальный тип* | *Имя вещественного типа*.
55. *Составной тип* = ["packed"] | *Неупакованный составной тип* | *Имя составного типа*.
56. *Ссылочный тип* = "↑" *Тип области* | *Имя ссылочного типа*.
57. *Ординальный тип* = *Перечисляемый тип* | *Диапазонный тип* | *Имя ординального типа*.
58. *Неупакованный составной тип* = *Массивовый тип* | *Записной тип* | *Множественный тип* | *Файловый тип*.
59. *Тип области* = *Имя типа*.
60. *Перечисляемый тип* = "("*Список имен*")".

61. *Диапазонный тип* = *Константа* ".." *Константа*.
62. *Массивовый тип* = "array" "[" *Тип индекса* {"," *Тип индекса*} "]" "of" *Тип компоненты*.
63. *Записной тип* = "record" *Список полей* "end".
64. *Множественный тип* = "set" "of" *Базовый тип*.
65. *Файловый тип* = "file" "of" *Тип компоненты*.
66. *Тип индекса* = *Ординальный тип*.
67. *Тип компоненты* = *Тип*.
68. *Базовый тип* = *Ординальный тип*.
69. *Тип результата* = *Имя ординального типа* | *Имя вещественного типа* | *Имя ссылочного типа*.
70. *Список полей* = [(*Фиксированная часть* [";" *Вариантная часть*]) *Вариантная часть*) [";"]].
71. *Фиксированная часть* = *Секция записи* {";" *Секция записи*}.
72. *Вариантная часть* = "case" *Селектор варианта* "of" *Вариант записи* {";" *Вариант записи*}.
73. *Секция записи* = *Список имен* ":" *Тип*.
74. *Селектор варианта* = [*Поле признака* ":"] *Тип признака*.
75. *Вариант записи* = *Константа* {"," *Константа*} ":" "(" *Список полей* ")".
76. *Тип признака* = *Имя*.
77. *Поле признака* = *Имя*.
78. *Константа* = [*Знак*] (*Число без знака* | *Имя константы*) | *Строка символов*.
79. *Выражение* = *Простое выражение* [*Оператор отношения* *Простое выражение*].
80. *Простое выражение* = [*Знак*] *Терм* {*Аддитивная операция* *Терм*}.
81. *Терм* = *Фактор* {*Мультипликативная операция* *Фактор*}.
82. *Фактор* = *Константа без знака* | *Имя границы* | *Переменная* | *Конструктор множества* | *Обозначение функции* | "not" *Фактор* | "(" *Выражение* ")".
83. *Операция отношения* = "=" | "<>" | "<" | "<=" | ">" | ">=" | "in".
84. *Аддитивная операция* = "+" | "—" | "or".
85. *Мультипликативная операция* = "*" | "/" | "div" | "mod" | "and".
86. *Константа без знака* = *Число без знака* | *Строка символов* | *Имя константы* | "nil".
87. *Обозначение функции* = *Имя функции* [*Список фактических параметров*].
88. *Переменная* = *Полная переменная* | *Переменная-компонента* | *Идентифицированная переменная* | *Буферная переменная*.
89. *Полная переменная* = *Имя переменной*.

90. *Переменная-компонента* = *Индексированная переменная* | *Обозначение поля*.
91. *Идентифицированная переменная* = *Ссылочная переменная* "↑".
92. *Буферная переменная* = *Переменная-файл* "↑".
93. *Индексированная переменная* = *Переменная-массив* "["*Индекс* {","*Индекс*}"]".
94. *Обозначение поля* = [*Переменная-запись* "."] *Имя поля*.
95. *Конструктор множества* = "["[*Описание элемента* {","*Описание элемента*}]"]".
96. *Описание элемента* = *Ординальное выражение* [".." *Ординальное выражение*].
97. *Список фактических параметров* = "(" *Фактический параметр* {"," *Фактический параметр*}")".
98. *Фактический параметр* = *Выражение* | *Переменная* | *Имя процедуры* | *Имя функции*.
99. *Список параметров вывода* = "("*Переменная-файл* | *Параметр вывода*) {"," *Параметр вывода* }")".
100. *Параметр вывода* = *Выражение* [":" *Целое выражение* [":" *Целое выражение*]].
101. *Переменная-массив* = *Переменная*.
102. *Переменная-запись* = *Переменная*.
103. *Переменная-файл* = *Переменная*.
104. *Ссылочная переменная* = *Переменная*.
105. *Целое выражение* = *Ординальное выражение*.
106. *Логическое выражение* = *Ординальное выражение*.
107. *Ординальное выражение* = *Выражение*.
108. *Имя ссылочного типа* = *Имя типа*.
109. *Имя составного типа* = *Имя типа*.
110. *Имя ординального типа* = *Имя типа*.
111. *Имя вещественного типа* = *Имя типа*.
112. *Имя константы* = *Имя*.
113. *Имя типа* = *Имя*.
114. *Имя переменной* = *Имя*.
115. *Имя поля* = *Имя*.
116. *Имя процедуры* = *Имя*.
117. *Имя функции* = *Имя*.
118. *Имя границы* = *Имя*.
119. *Число без знака* = *Целое без знака* | *Вещественное без знака*.
120. *Список имен* = *Имя* [","*Имя*].
121. *Имя* = *Буква* [*Буква* | *Цифра*].
122. *Директива* = *Буква* [*Буква* | *Цифра*].
123. *Метка* = *Последовательность цифр*.
124. *Целое без знака* = *Последовательность цифр*.

125. *Вещественное без знака* = *Целое без знака* "." *Последовательность цифр* ["e" *Порядок*] | *Целое без знака* "e" *Порядок*.
126. *Порядок* = [*Знак*] *Целое без знака*.
127. *Знак* = "+" | "−".
128. *Строка символов* = "'" *Элемент строки* {*Элемент строки*} "'".
129. *Последовательность цифр* = *Цифра* {*Цифра*}.
130. *Буква* = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
131. *Цифра* = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0".
132. *Элемент строки* = "'" *Любой символ кроме апострофа*.

АЛФАВИТНЫЙ СПИСОК МЕТАИМЕН КОНСТРУКЦИЙ СО ССЫЛКАМИ

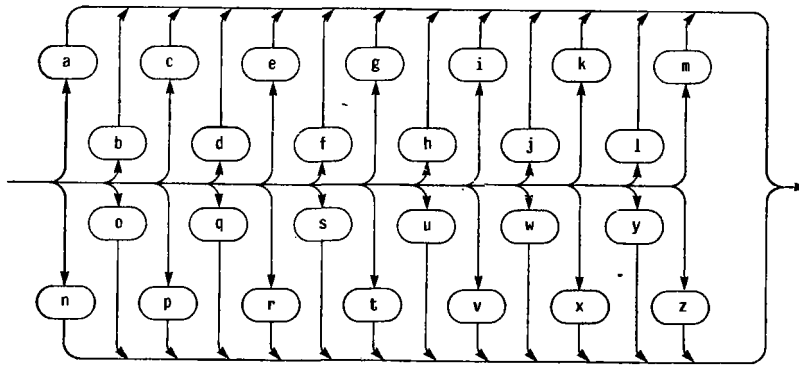
Параграф описания		**Номер правила**
8.	*Аддитивная операция*	84
6.2.3.	*Базовый тип*	68
10.1.	*Блок*	4
4.	*Буква*	130
7.4.	*Буферная переменная*	92
9.2.2.2.	*Вариант*	49
6.2.2.	*Вариант записи*	75
6.2.2.	*Вариантная часть*	72
4.	*Вещественное без знака*	125
8.	*Выражение*	79
9.2.2.	*Выбирающий оператор*	35
6.1.3.	*Диапазонный тип*	61
4.	*Директива*	122
13.	*Заголовок программы*	2
11.1.	*Заголовок процедуры*	16
11.2.	*Заголовок функции*	18
6.2.2.	*Записной тип*	63
4.	*Знак*	127
11.1.	*Идентификация процедуры*	17
11.2.	*Идентификация функции*	19
7.3.	*Идентифицированная переменная*	91
4.	*Имя*	121
6.1.	*Имя вещественного типа*	111
11.3.1.1.	*Имя границы*	118
5.	*Имя константы*	112

7.2.1.	*Переменная-массив*	101
7.4.	*Переменная-файл*	103
6.1.1.	*Перечисляемый тип*	60
6.2.2.	*Поле признака*	77
7.1.	*Полная переменная*	89
4.	*Порядок*	126
9.2.	*Последовательность операторов*	31
4.	*Последовательность цифр*	129
13.	*Программа*	1
8.	*Простое выражение*	80
9.1.	*Простой оператор*	33
6.1.	*Простой тип*	54
9.1.	*Пустой оператор*	37
9.	*Раздел операторов*	10
10.1.1.	*Раздел описания меток*	5
7.	*Раздел описания переменных*	8
11.	*Раздел описания процедур и функций*	9
5.	*Раздел определения констант*	6
6.	*Раздел определения типов*	7
6.2.2.	*Селектор варианта*	74
6.2.2.	*Секция записи*	73
11.3.1.	*Секция формальных параметров*	21
9.2.	*Сложный оператор*	34
9.2.1.	*Составной оператор*	30
6.2.	*Составной тип*	55
12.3.	*Список параметров вывода*	99
11.2.1.1.	*Спецификация параметров-значений*	22
11.3.1.1.	*Спецификация параметров-переменных*	23
11.3.1.2.	*Спецификация процедурального параметра*	24
11.3.1.1.	*Спецификация типа индекса*	29
11.3.1.2.	*Спецификация функционального параметра*	25
6.1.1.	*Список имен*	120
13.	*Список параметров программы*	3
9.2.4.	*Список переменных-записей*	47
6.2.2.	*Список полей*	70
11.3.2.	*Список фактических параметров*	97
11.3.1.	*Список формальных параметров*	20
7.3.	*Ссылочная переменная*	104
6.3.	*Ссылочный тип*	56
4.	*Строка символов*	128
11.3.1.1.	*Схема неупакованного совмещаемого массива*	28
11.3.1.1.	*Схема упакованного совмещаемого массива*	27

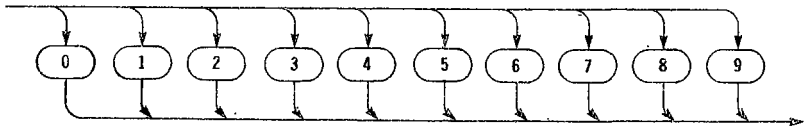
СИНТАКСИЧЕСКИЕ ДИАГРАММЫ

Диаграммы для метаимен *Буква, Цифра, Имя, Директива, Целое без знака, Число без знака* и *Строка символов* описывают образование из символов этих лексем. Другие же диаграммы описывают образование из лексем синтаксических конструкций.

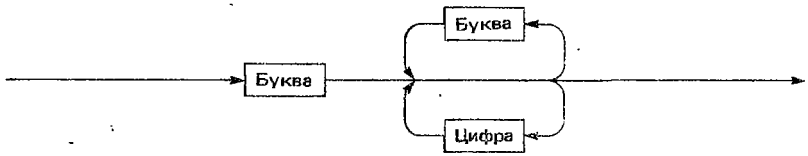
Буква



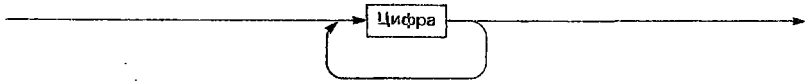
Цифра



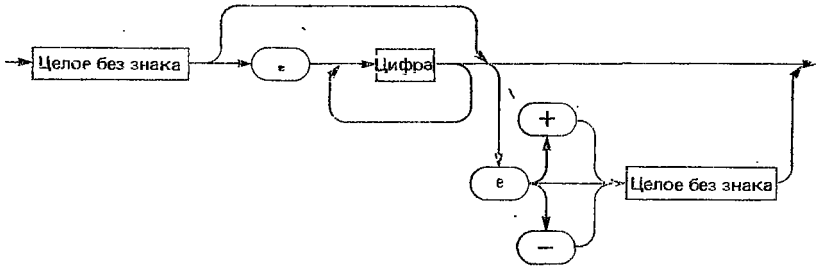
Имя и Директива



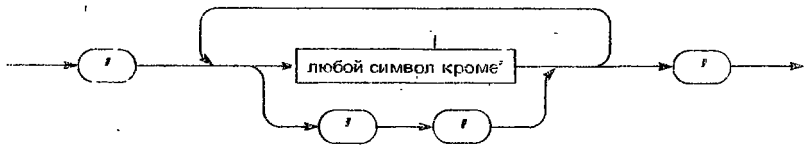
Целое без знака



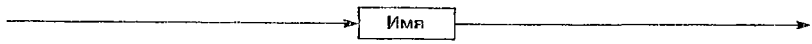
Число без знака



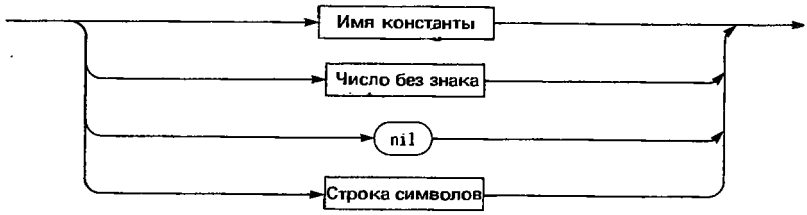
Строка символов



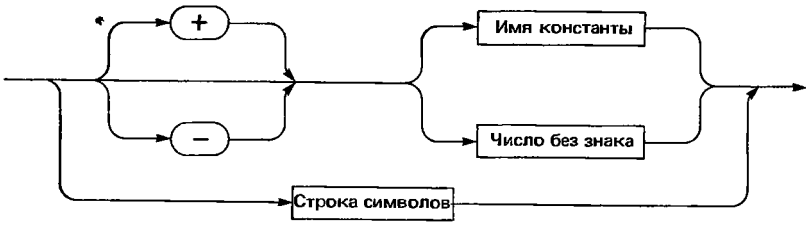
Имя константы, Имя переменной, Имя поля, Имя границы, Имя типа, Имя процедуры и Имя функции



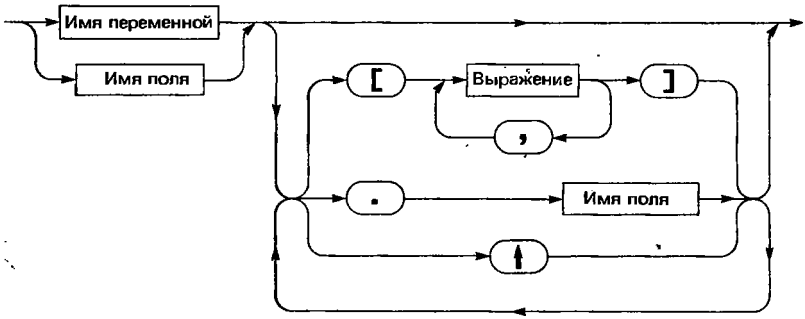
Константа без знака



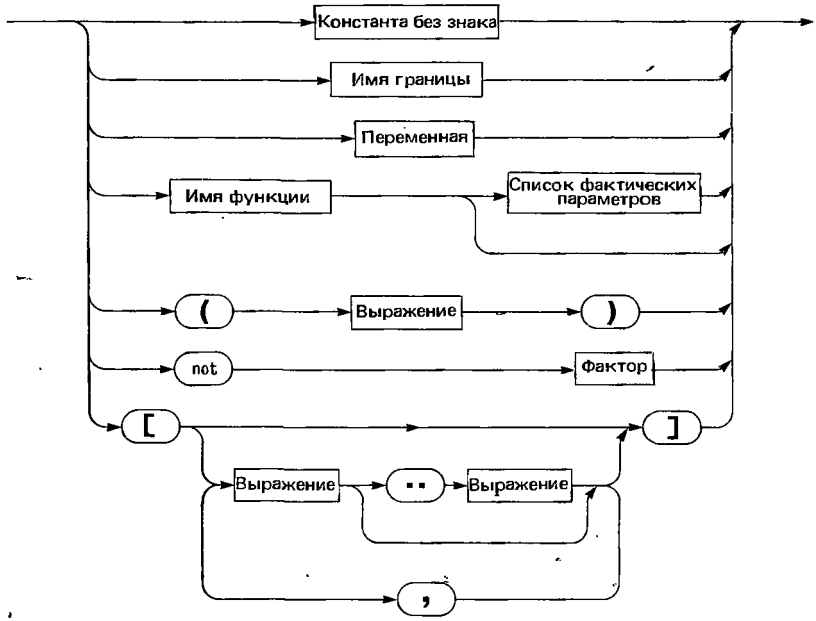
Константа



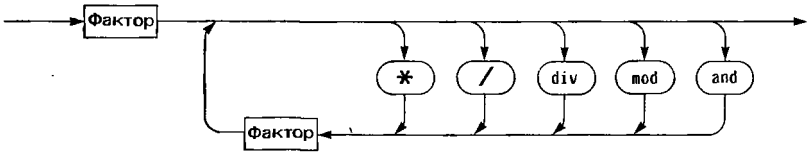
Переменная



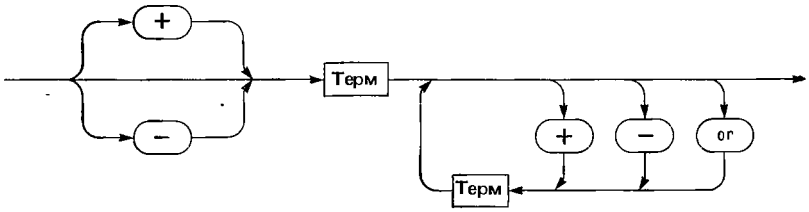
Фактор



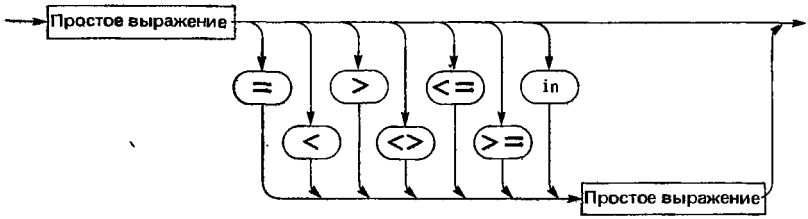
Терм



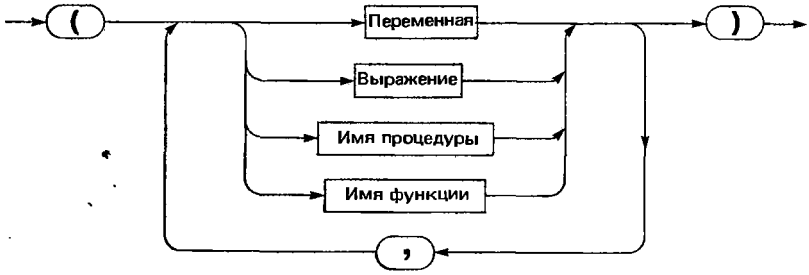
Простое выражение



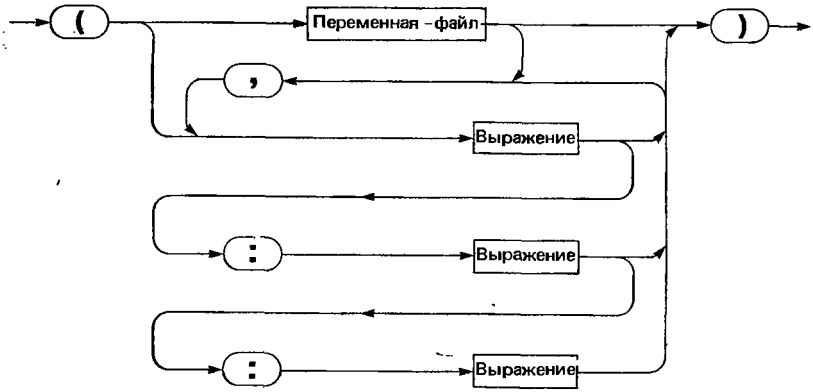
Выражение



Список фактических параметров

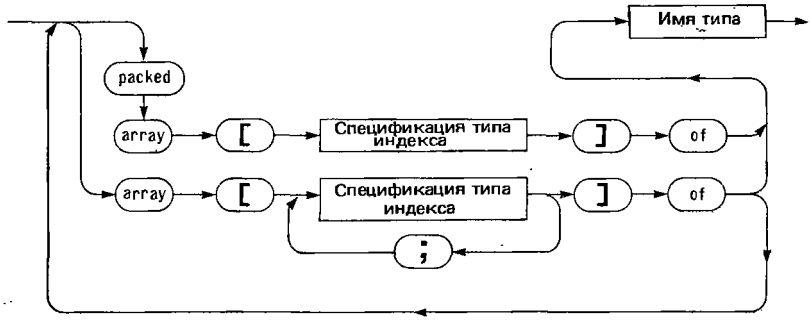


Список параметров вывода

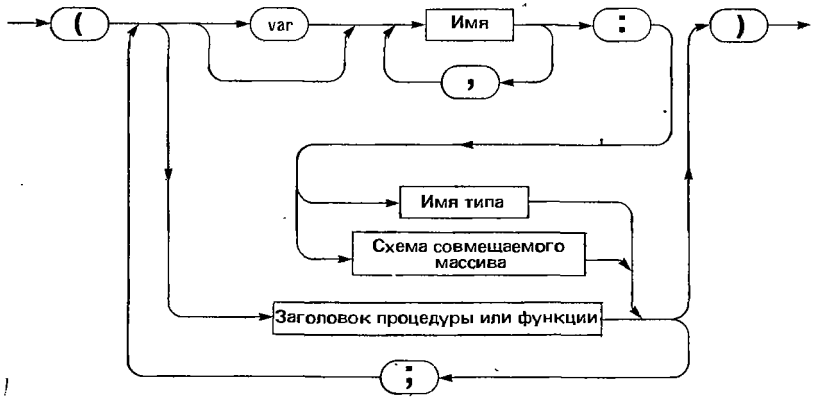


Спецификация типа индекса

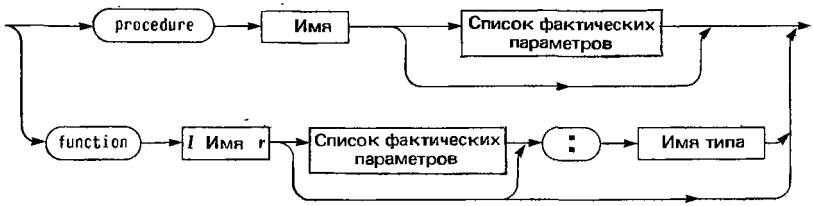
Схема совмещаемого массива



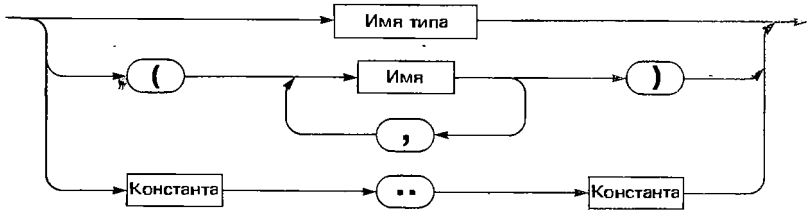
Список формальных параметров



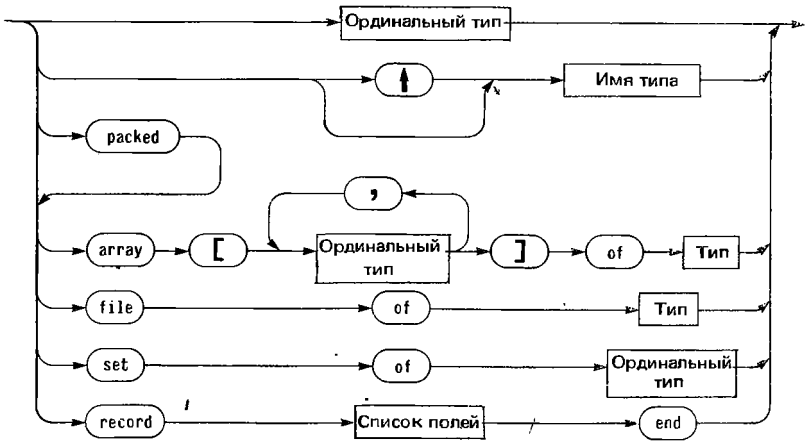
Заголовок процедуры или функции



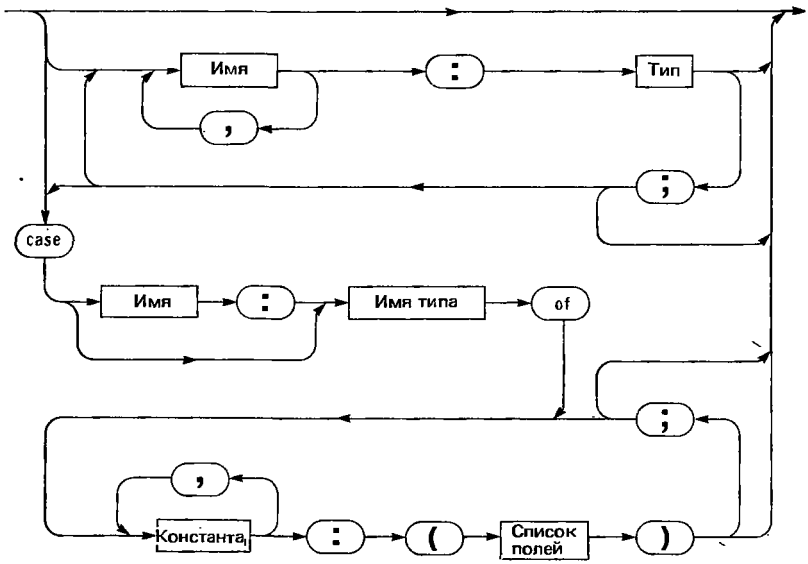
Ординальный тип



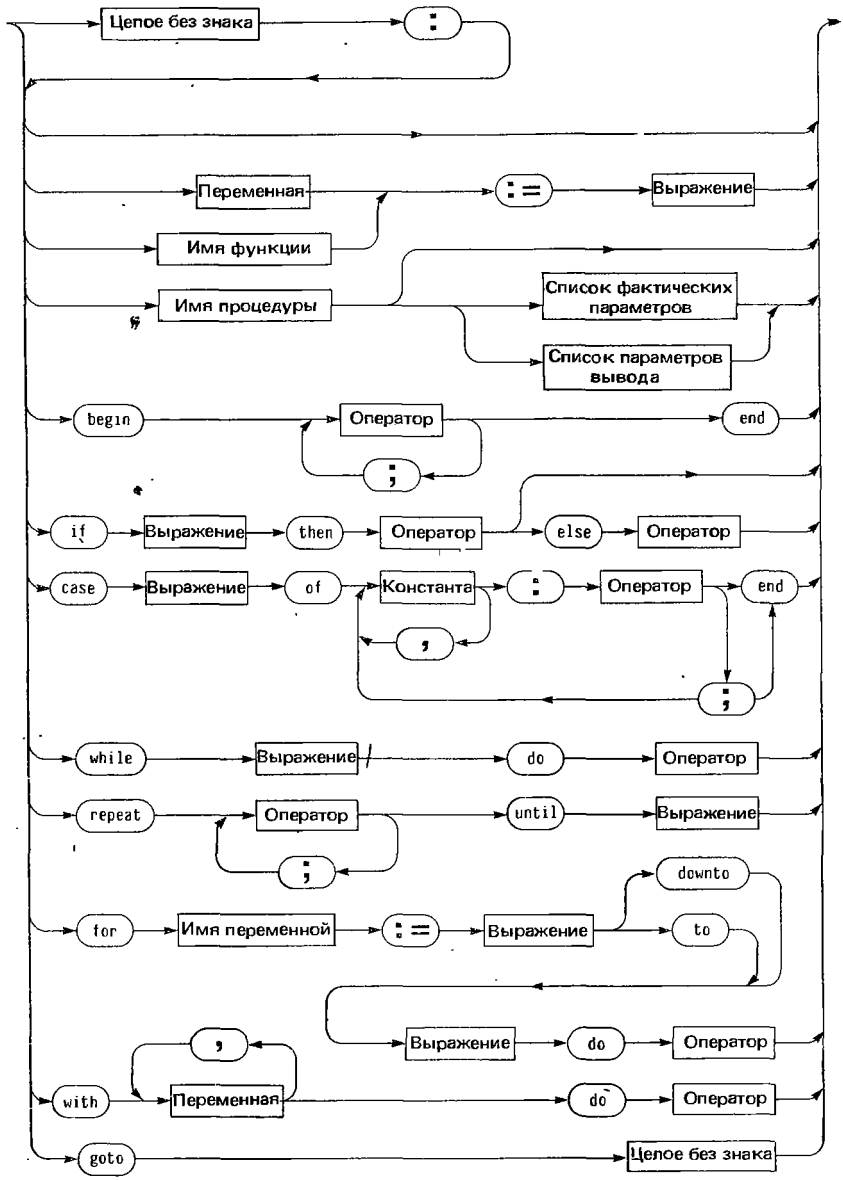
Тип:



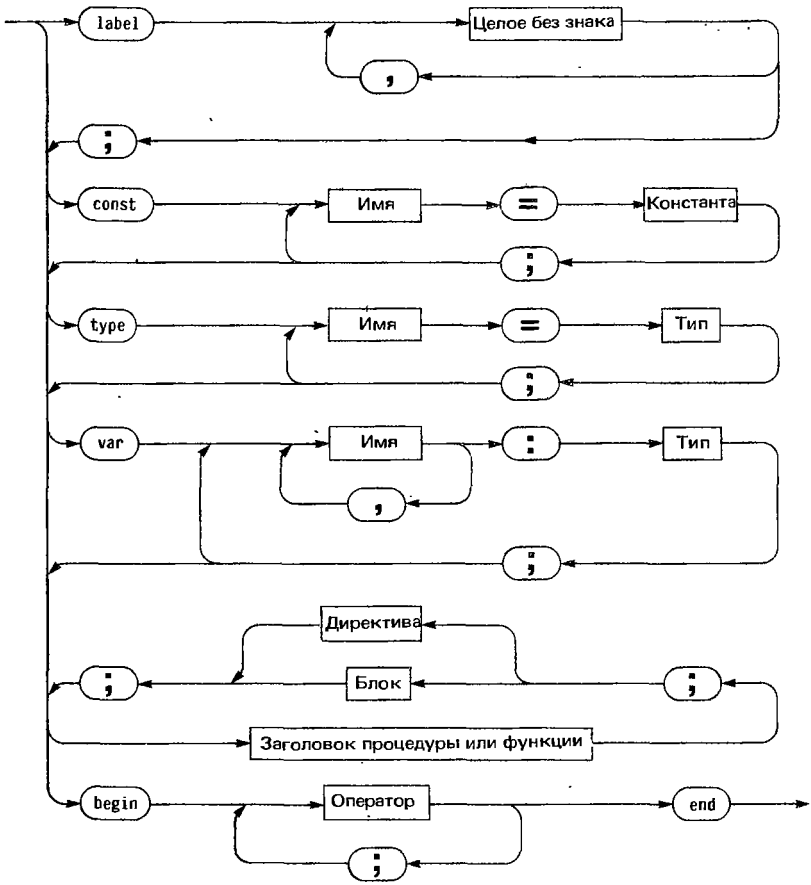
Список полей



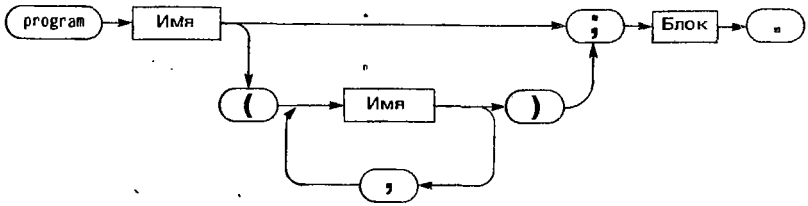
Оператор


Целое без знака
:
Переменная
:=
Выражение
Имя функции
Имя процедуры
Список фактических параметров
Список параметров вывода
begin
Оператор
end
;
if
Выражение
then
Оператор
else
Оператор
case
Выражение
of
Константа
:
Оператор
end
,
;
while
Выражение
do
Оператор
repeat
Оператор
until
Выражение
;
for
Имя переменной
:=
Выражение
downto
to
Выражение
do
Оператор
,
with
Переменная
do
Оператор
goto
Целое без знака

Блок



Программа



ПРИЛОЖЕНИЕ 5

ИЗМЕНЕНИЯ В «РУКОВОДСТВЕ ДЛЯ ПОЛЬЗОВАТЕЛЯ» И «ОПИСАНИИ ЯЗЫКА», ОБУСЛОВЛЕННЫЕ СТАНДАРТОМ ИСО 7185

В этом приложении приводится не полный обзор технических изменений, сделанных при подготовке третьего издания книги, согласованного со стандартом ИСО. Этот материал может оказаться полезным читателям, знакомым с предыдущими изданиями.

Описание 3: нотация, терминология и лексика

Вместо БНФ (BNF) используется РБНФ (EBNF).

Дается определение понятий *ошибки, зависимости от реализации* (implementation-dependent), *определения при реализации* (implementation-defined), *расширения* (extension) и *стандартного Паскаля*. Эти определения используются на протяжении всего «Описания».

Описание 4: имена, числа и строки

В описании синтаксиса понятие «ограничитель» (delimiter) заменяется на понятие «разделитель» (separator).

Добавляется символ "..".

Вводится альтернативное представление специальных символов "[", "]" и "↑".

Изменяется синтаксис примечаний: не допускаются вложенные примечания.

В именах приобретают значение все их символы.

Появляется новая категория слов: директивы.

Описание 5: константы

В определение включается константа MaxInt.

Описание 6: типы

Скалярные типы заменяются на ординальные и вещественные, что приводит к упрощению определений функций succ, pred и ord и понятий «индексаций массива», «селектор варианта», «диапазон» и «базовый тип множества».

Совместимость типов теперь определяется как совместимость «по имени» (name compatibility).

Вводится концепция «совместимости при присваивании» (assignment compatibility) и «присваиваемых типов» (assignable type).

Появляется специфическое семантическое толкование «упакованных составных типов».

В полном синтаксисе для записных типов разрешается использование символа ";".

Метки вариантов в вариантных записях называются константами вариантов.

В записных типах в вариантной части теперь требуется полная спецификация.

Для файловых типов определяются режимы формирования (generation) и просмотра (inspection mode).

Тип text теперь не эквивалентен (упакованному) файлу из символов.

Типы компонент файловых типов не могут быть в свою очередь файловыми типами или типами, содержащими файловые типы.

Для ссылочных типов вводятся типы областей (domain types).

Описание 7: переменные

Вводится понятие *неопределенной* (undefined) и *полностью неопределенной* (totally undefined) переменных.

Имена Input и Output, если они используются, относятся к неявно описанным параметрам программы, представляющим собой текстовые файлы.

Описание 8: выражения

В фактор (множитель) теперь может входить имя границы для совмещаемого массива-параметра.

Порядок вычисления выражений декларируется как зависящий от реализации.

Изменено определение операции mod.

Конструктор любого множественного типа теперь может быть и упакованным, и неупакованным.

Описание 9: операторы

Ужесточаются правила, связанные с доступностью меток через оператор перехода.

Метка оператора варианта называется индексом варианта (case constant).

Управляющая переменная оператора цикла с шагом становится просто локальной переменной.

В оператор цикла с шагом добавляются некоторые ограничения, и его действие определяется более строго.

Описание 10: блоки, области действия и активации

Определяются понятия *точки программы* (program-point), *точки активации* (activation-point), *области действия для определения или описания (введения)* меток и имен.

Описание 11: процедуры и функции

Вводятся директивы, относящиеся к процедурам и функциям; директива forward становится стандартной.

Добавляются в качестве параметров совмещаемые (conformant) массивы; вводится концепция *совмещаемости* (conformability) и появляются совмещаемые типы.

Для формальных процедуральных или функциональных параметров (т. е. процедур или функций, фигурирующих как формальные параметры) теперь требуется полная спецификация всего списка параметров. Вводится концепция *конгруэнтности списков параметров.*

Использование поля признака в качестве фактического параметра-переменной не допускается.

Изменяется спецификация массива-параметра как упакованного или неупакованного.

Теперь строго определяются функции и процедуры, работающие с файлами, состояния переменной-файла и буферной переменной.

Описание 12: текстовые файлы, ввод и вывод

Вводится стандартная процедура Page, у нее допускается параметр — файл. Действие процедуры изменяется.

Для списка фактических параметров процедур write и writeln вводится особый синтаксис — конструкций «Список параметров вывода» (Write parameter list).

Точно определяется смысл «весового» (width) поля при форматировании в процедурах write и writeln.

Описание 13: программы

Для программы допускаются параметры; определяется их смысл.

Описание 14: согласованность со стандартом ИСО 7185

Дается определение *согласованной программы* и *согласованного процессора.*

Объясняются требования согласованности (compliance) со стандартом ИСО Паскаля.

ПРИЛОЖЕНИЕ 6

ПРИМЕРЫ ПРОГРАММ

В этом приложении мы приводим два примера: первый иллюстрирует процесс разработки программы с помощью метода пошагового уточнения [2], а второй дает модель переносимой программы.

Пример 1: Program IsItAPalindrome

Необходимо написать программу, которая будет отыскивать в интервале от 1 до 100 все целые числа, квадраты которых представляют собой палиндромы. Например, 11 в квадрате составит 121 — а это палиндром.

Палиндромом называется в некотором алфавите строка символов, одинаково читающаяся в любом направлении. Если игнорировать пробелы и знаки пунктуации, то палиндромами являются следующие английские фразы:

"radar"
"a man, a plan, a canal, Panama"
"Doc, note, I dissent! A fast never prevents a fatness: I diet on cod".

Пример 1, шаг 1:

```
program IsItAPalindrome(Output);

begin
   FindAllIntegersFrom1To100WhoseSquaresArePalindromes
end { IsItAPalindrome }
```

Пример 1, шаг 2:

```
program IsItAPalindrome(Output);

   { Поиск всех целых от 1 до 100, квадраты которых — палиндромы. }

   const
      Maximum = 100;
```

```
   type
      IntRange = 1..Maximum;

   var
      N: IntRange;

begin
   for N := 1 to Maximum do
      if Palindrome(Sqr(N)) then
         Writeln(N, ' squared is a palindrome.')
end { IsItAPalindrome }
```

Пример 1, шаг 3:

```
program IsItAPalindrome(Output);

   { Поиск всех целых от 1 до 100, квадраты которых — палиндромы. }

   const
      Maximum = 100;

   type
      IntRange = 1..Maximum;

   var
      N: IntRange;

   function Palindrome(Square: Integer): Boolean;

      var
         NPlaces = 1..5 { 5 = Log10(Sqr(Maximum)) + 1 }

   begin { Palindrome }
      CrackDigits;
      Palindrome := CheckSymmetry(1, NPlaces)
   end { Palindrome };

begin
   for N := 1 to Maximum do
      if Palindrome(Sqr(N)) then
         Writeln(N, ' squared is a palindrome.')
end { IsItAPalindrome } .
```

Пример 1, шаг 4:

```
program IsItAPalindrome(Output);

  { Поиск всех целых от 1 до 100, квадраты которых — палиндромы. }

  const
    Maximum = 100;

  type
    IntRange = 1..Maximum;

  var
    N: IntRange;

  function Palindrome(Square: Integer): Boolean;
  const
    Places = 5 { = Trunc(Log10(Sqr(Maximum))) + 1 };

  type
    NPlaces = 1..Places;
    SingleDigit = 0..9;
    DigitVec = array [NPlaces] of SingleDigit;

  var
    Digits: DigitVec;
    Size: NPlaces;

  procedure CrackDigits;
  begin
    Size := 1;
    while Square > 9 do
      begin
        Digits[Size] := Square mod 10;
        Square := Square div 10;
        Size := Size + 1
      end;
    Digits[Size] := Square
  end { CrackDigits };

  function CheckSymmetry(Left, Right: NPlaces): Boolean;
  begin
```

```
      if Left >= Right then CheckSymmetry := true
      else
         if Digits[Left] = Digits[Right] then
            CheckSymmetry := CheckSymmetry(Left + 1, Right - 1)
         else CheckSymmetry := false
   end { CheckSymmetry };
      begin { Palindrome }
         CrackDigits;
         Palindrome := CheckSymmetry(1, Size)
      end { Palindrome };

   begin
      for N := 1 to Maximum do
         if Palindrome(Sqr(N)) then
            Writeln(N, ' squared is a palindrome.')
   end { IsItAPalindrome } .
```

Пример 2: Procedure ReadRadi x Representation

Обобщенная процедура чтения целых чисел, записанных в позиционной системе с любым основанием от 2 до 16.

```
type Radix = 2..16;

procedure ReadRadixRepresentation
          (var F: Text;      { содержит представление  }
           var E: Boolean;   { указывает на ошибку }
           var X: Integer;   { если ошибок нет, то-результат  }
               R: Radix      {основание представления}
          );
```

{ ReadRadixRepresentation рассчитана на то, что текстовый файл F готов для чтения и в нем находится последовательность суперцифр, образующая целое число по основанию R. Суперцифрами являются:

'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'

Вместо прописных букв можно использовать строчные.

Параметр E указывает, что встретилась одна из следующих ошибок:

(1) Текстовый файл не поставлен в начало последовательности суперцифр.

(2) Последовательность задает число, превосходящее Maxint.

(3) В последовательности суперцифр есть цифра, не относящаяся к основанию R. }

```
   type
      DigitRange = 0..15;

   var
      D: DigitRange;
      V: Boolean;
      S: 0..Maxint;

   procedure ConvertExtendedDigit(C: Char; var V: Boolean;
                                  var D: DigitRange);
      { ConvertExtendedDigit определяет, относится ли C к
        суперцифрам, V указывает допустимость C и если V = true,
        заносит в D числовое значение этой суперцифры }

   begin { ConvertExtendedDigit }
      V := C in [ '0'..'9','a','b','c','d','e','f',
                  'A','B','C','D','E','F' ];
      if V then
         case C of
            '0': D := 0;  '1': D := 1;  '2': D := 2;  '3': D :=  3;
            '4': D := 4;  '5': D := 5;  '6': D := 6;  '7': D :=  7;
            '8': D := 8;  '9': D := 9;
            'A','a': D := 10;  'B','b': D := 11;  'C','c': D := 12;
            'D','d': D := 13;  'E','e': D := 14'  'F','f': D := 15;
         end
   end { ConvertExtendedDigit };

begin { ReadRadixRepresentation }
   E := true;
   ConvertExtendedDigit(F↑,V,D);
   if V then
      begin
         E := false;  S := 0,
         repeat
            if D < R then
               if (Maxint - D) div R >= S then
                  begin
                     S := S * R + D;
                     Get(F);
                     ConvertExtendedDigit(F↑,V,D);
                  end
               else E := true
            else E := true
         until E or not V;
         if not E then X := S
      end
end { ReadRadixRepresentation } .
```

ПРИЛОЖЕНИЕ 7

МНОЖЕСТВО СИМВОЛОВ ASCII

Множество символов ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией) представляет собой принятый в США вариант международного кода, известного под названием кода ИСО. Он определяет кодировку 128 символов. Предусматривается и существование национального варианта двенадцати таких символов, как, например, символ стоимости — $. Основное множество из 128 символов разбивается на 95 «графических» символов, которые могут быть напечатаны, и 33 «управляющих» символа — с их помощью управляют устройствами. Управляющий символ «возврат» (backspace) используется специально для «наложения» одного символа на другой, это необходимо в некоторых языках.

*Вот эти 33 управляющих символа**:

ACK	Подтверждение (Acknowledge)
BEL	Звонок (Bell)
BS	Возврат на шаг (Backspace)
CAN	Аннулирование (Cancel)
CR	Возврат каретки (Carriage Return)
DC1	Управление устройством 1 (Device Control 1)
DC2	Управление устройством 2 (Device Control 2)
DC3	Управление устройством 3 (Device Control 3)
DC4	Управление устройством 4 (Device Control 4)
DEL	Вычеркивание (Delete)
DLE	Авторегистр 1 (Data Link Escape)
EM	Конец носителя (End of Medium)
ENQ	Кто там? (Enquiry)
EOT	Конец передачи (End of Transmission)
ESC	Авторегистр 2 (Escape)
ETB	Конец блока (End of Transmission Block)
ETX	Конец текста (End of Text)
FF	Перевод формата (Form Feed)
FS	Разделитель файла (File Separator)
GS	Разделитель группы (Group Separator)

* Русские названия управляющих символов (кроме SUB) взяты из ГОСТ 13052—67. — *Примеч. пер.*

HT	Горизонтальная табуляция (Horizontal Tab)
LF	Перевод строки (Line Feed)
NAK	Отрицание (Negative Acknowledge)
NUL	Пусто (Null)
RS	Разделитель записи (Record Separator)
SI	Латинский регистр (Shift In)
SO	Национальный регистр (Shift Out)
SOH	Начало заголовка (Start of Heading)
STX	Начало текста (Start of Text)
SUB	Подстановка (Substitute)
SYN	Синхронизация (Synchronous Idle)
US	Разделитель элемента записи (Unit Separator)
VT	Вертикальная табуляция (Vertical Tab)

А теперь все 128 символов:

	00	16	32	48	64	80	96	112
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	|
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

Семиразрядный код для символа представляет собой сумму номера строки и номера столбца. Например, код для символа G — 7 + 64 = 71.