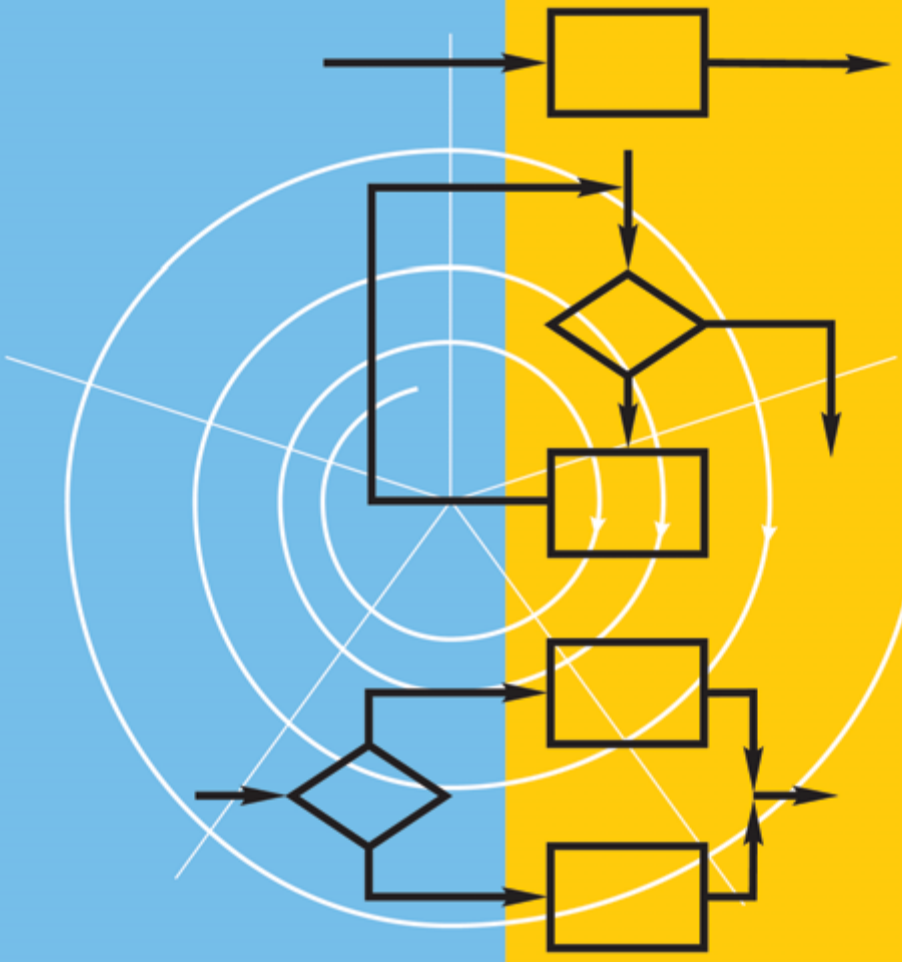
Л. ЗАЛОГОВА

РАЗРАБОТКА ПАСКАЛЬ-КОМПИЛЯТОРА



Л. ЗАЛОГОВА

РАЗРАБОТКА ПАСКАЛЬ-КОМПИЛЯТОРА

3-е издание (электронное)

Москва
БИНОМ. Лаборатория знаний
2014

УДК 004.4’42
ББК 32.973.26-018.2
З-24

Залогова Л. А.

З-24 Разработка Паскаль-компилятора [Электронный ресурс] / Л. А. Залогова. — 3-е изд. (эл.). — Электрон. текстовые дан. (1 файл pdf : 186 с.). — М. : БИНОМ. Лаборатория знаний, 2014. — Систем. требования: Adobe Reader XI ; экран 10".

ISBN 978-5-9963-2526-9

В книге излагается структура компилятора, основные принципы построения всех его основных блоков — лексического, синтаксического и семантического анализаторов, а также генератора кода. Методы компиляции программ на Паскале описаны на языке С.

Для студентов и специалистов, занимающихся созданием программного обеспечения, а также для всех, желающих создать компилятор с своего собственного языка программирования.

УДК 004.4’42
ББК 32.973.26-018.2

Деривативное электронное издание на основе печатного аналога: Разработка Паскаль-компилятора / Л. А. Залогова. — М. : БИНОМ. Лаборатория знаний, 2007. — 183 с. : ил.

ISBN 978-5-9963-2526-9

Оглавление

Введение

Компиляторы — это важнейшая область исследований, связанных с программным обеспечением, и одновременно — одна из главных составляющих инструментария разработчиков программ. Без компиляторов программистам пришлось бы писать программы непосредственно в машинных кодах, единственно понятных компьютеру. Компиляторы же позволяют создавать программы на языках, понятных и удобных для человека, а затем переводят (транслируют) их в машинные коды.

Цели этой книги: рассмотреть типовую структуру компилятора, а именно представить компилятор как совокупность логически взаимосвязанных модулей; определить взаимодействие между этими модулями и изучить принципы их построения; и наконец, используя метод пошаговой детализации, описать основные функции отдельных модулей.

Чтобы продемонстрировать на практике создание собственного компилятора, в книге описываются методы компиляции Паскаль-программ средствами языка С. Это позволит читателям, знакомым с наиболее распространенными языками программирования Паскаль и С, самим научиться писать компиляторы. Кроме того, для понимания материала книги читатель должен знать способы представления различных информационных структур (данных) в памяти компьютера, а также основные алгоритмы работы с ними.

В книге излагается один из возможных методов создания Паскаль-компилятора. Другие, альтернативные варианты здесь не рассматриваются.

Материал, изложенный в книге, используется автором в течение нескольких лет при чтении курса лекций и проведении практических занятий на механико-математическом факультете Пермского государственного университета. В рамках этого курса каждый студент должен написать, отладить и выполнить тестирование компилятора на примере некоторого фрагмента на языке Паскаль. Только в этом случае действительно становится понятным, как работает компилятор. Как показывает практика, знания и навыки, полученные в результате изучения этого курса, позволяют создать компилятор для своего собственного языка программирования.

Глава 1

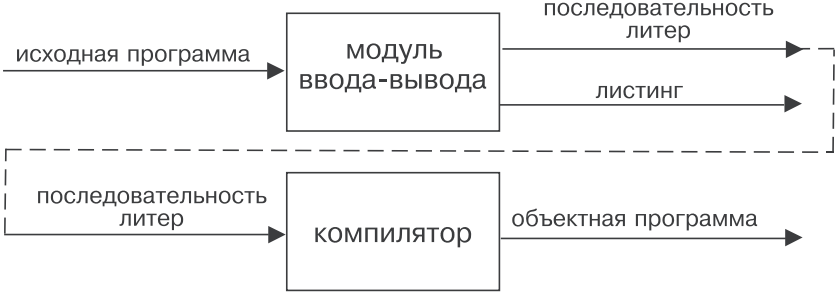
Структура компилятора

Компилятор — это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке. Обычно компилятор также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках.

Разработка программного обеспечения (ПО) подразумевает **модульность** и хорошую **структурированность** программ. Учитывая это, представим компилятор как совокупность логически взаимосвязанных модулей, определим взаимодействие между ними и, используя **метод пошаговой детализации**, опишем основные функции отдельных модулей на языке С.

Следуя определению, изобразим компилятор в виде схемы:

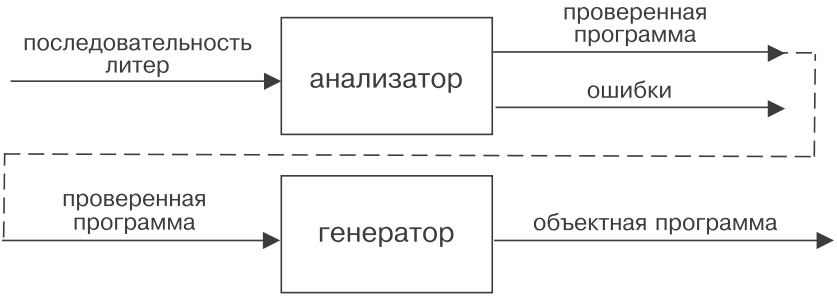
При вводе исходной программы и получении листинга мы имеем дело с конкретными устройствами ввода-вывода (клавиатура, дисплей, магнитные диски). Чтобы легко адаптировать компилятор к различным внешним устройствам конкретной машины, отделим все действия по вводу-выводу данных от собственно процесса компиляции:



Работа компилятора включает в себя два основных этапа:

1) **анализ** — определение правильности исходной программы и формирование (в случае необходимости) сообщений об ошибках;
2) **синтез** — генерация объектной программы; этот этап выполняется для программ, не содержащих ошибок.

Таким образом, собственно компилятор разбивается на составляющие модули:



Одно из достоинств компилятора заключается в возможности генерировать объектную программу для компьютеров с различной архитектурой и различных операционных систем. Однако сам компилятор представляет собой машинно-зависимую программу, так как результат его работы определяется архитектурой конкретного компьютера, а именно представлением данных, кодами операций, форматами машинных команд, способами адресации и т. д. Поэтому уже на ранних стадиях проектирования компилятора в нем выделяют машинно-зависимые и машинно-независимые части. В этом случае работа при переносе компилятора с одной машины на другую существенно облегчается. При этом **анализатор представляет собой машинно-независимую часть компилятора**, а **генератор**, который отображает машинно-независимое промежуточное представление исходной программы на реальную ЭВМ и должен переписываться для каждой новой машины, — это **машинно-зависимая часть компилятора**.

При проверке правильности программы используется полное описание *синтаксиса языка программирования*. Для задания синтаксиса широко применяются **формальные правила** — формы Бэкуса—Наура, а также синтаксические диаграммы.

Например, описание раздела объявления переменных в виде форм Бэкуса—Наура выглядит так:

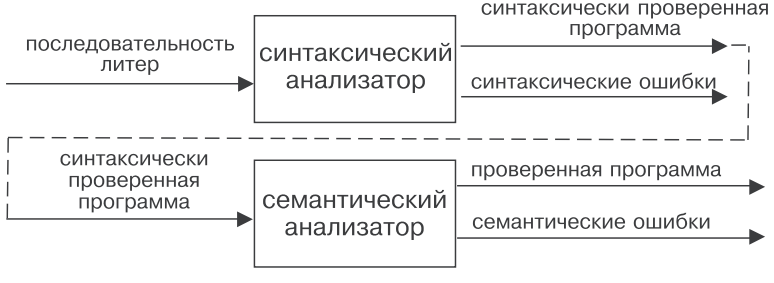
```
<раздел переменных> ::=
   var <описание однотипных переменных> ;
    {<описание однотипных переменных> ; }| < пусто>
<описание однотипных переменных> ::=
                          <имя> { , <имя>} : <тип>
```

В соответствии с этим описанием, следующий фрагмент программы на Паскале:

```
var   name1, name1, name2 : integer;
   name1, name2, name2: real;
```

не содержит ошибок. Причина заключается в том, что, с точки зрения форм Бэкуса—Наура, конкретное представление имени не имеет значения. Поэтому дополнительно к формальным правилам синтаксис языков программирования должен описываться *неформально* — с помощью естественного языка. В нашем примере это неформальное правило формулируется так: «в любой области действия без внутренних по отношению к ней областей никакой идентификатор не может быть описан более одного раза».

Учитывая особенности описания синтаксиса языков программирования, разделим анализатор на два модуля:

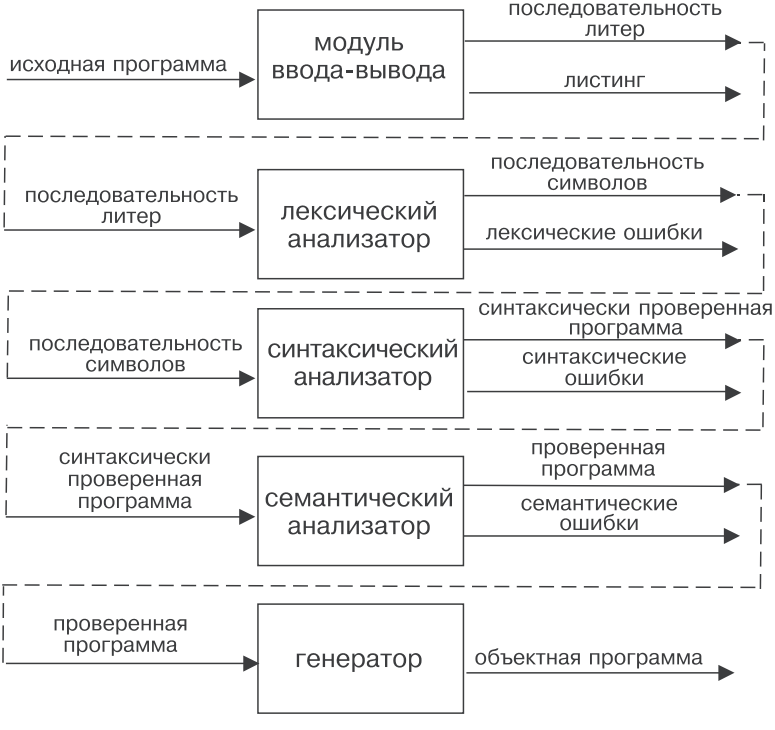


Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам. Назначение же **семантического анализатора** состоит в том, чтобы выяснить, не нарушены ли неформальные правила описания языка.

Дальнейшее разбиение на модули обычно выполняется внутри синтаксического анализатора. Первый модуль **(сканер)** просматривает текст (последовательность литер) исходной программы и строит **символы (лексемы)** — идентификаторы, ключевые слова, разделители, числа. Фактически сканер осуществляет простой *лексический анализ* исходной программы, поэтому его называют **лексическим анализатором.**

Второй модуль **(синтаксический анализатор)** выполняет *синтаксический анализ* последовательности символов. На этом этапе символы рассматриваются как *неделимые*, и их представление как последовательности литер несущественно.

Итак, в результате мы получили следующую структуру компилятора:



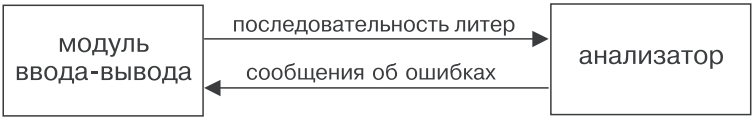
Все указанные на схеме *фазы компиляции* могут работать последовательно или параллельно, но с определенной взаимной синхронизацией. Если какая-нибудь фаза требует чтения исходного текста или результата его трансляции на некоторый внутренний язык, то это обычно называется *проходом*. В некоторых случаях программа полностью компилируется за один проход; при этом обычное ограничение для *однопроходных компиляторов* заключается в том, что все идентификаторы должны быть описаны до их использования. В ряде случаев необходимо иметь несколько проходов, и критерии, определяющие выбор количества проходов, могут быть самыми разнообразными.

Глава 2

Модуль ввода-вывода

2.1. Взаимодействие между модулем ввода-вывода и анализатором

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их анализатору. **Анализатор** проверяет, удовлетворяет ли эта последовательность литер правилам описания языка, и формирует (в случае необходимости) сообщения об ошибках. Такое взаимодействие между модулем ввода-вывода и анализатором можно представить в виде схемы:



Будем считать, что в результате очередного обращения к модулю ввода-вывода анализатор получает текущую литеру в переменной:

```
char ch;
```

Чтобы напечатать сообщение об ошибке, анализатор должен передать модулю ввода-вывода причину и местоположение ошибки. Так как она может встретиться в любом месте исходной программы, анализатору необходимо знать *координаты всех литер во входном потоке*. Поэтому модуль ввода-вывода должен формировать номер строки и номер позиции в строке для каждой литеры:

```
struct textposition positionnow;
```

Определим структуру **textposition**:

```
struct textposition
  {
    unsigned linenumber; /*номер строки */
    unsigned charnumber; /*номер позиции
                           в строке */
  };
```

Анализатор выявляет максимально возможное количество ошибок за один просмотр исходной программы. Информация об ошибках заносится в *таблицу ошибок*, каждый элемент которой обязательно содержит код и местоположение ошибки. Анализатор запоминает информацию об ошибках в таблице ошибок в результате обращения к функции:

```
error ( unsigned errorcode ,    /* код ошибки */
        textposition position   /* местоположение
                                        ошибки */ )
```

Модуль ввода-вывода использует содержимое этой таблицы для печати сообщений об ошибках при формировании листинга.

2.2. Программирование модуля ввода-вывода

Структура модуля ввода-вывода (функция **nextch**) может быть представлена следующим образом:

```
nextch( )
 { if ( текущая литера является последней
        литерой строки )
    { напечатать текущую строку;
      if ( в текущей строке обнаружены ошибки )
         напечатать соответствующие сообщения;
      прочитать следующую строку;
    }
   установить в качестве текущей
   следующую литеру и запомнить ее координаты;
 }
```

Будем считать, что максимальная длина строки исходной программы определяется константой **MAXLINE**. Так как исходная программа считывается построчно, буфер ввода-вывода опишем как массив:

```
char line [MAXLINE]
```

Длина строки, считываемой с внешнего запоминающего устройства, может быть меньше размера буфера, поэтому введем переменную:

```
short LastInLine;
```

значение которой — это количество литер в текущей строке.

Первая строка программы пользователя должна быть считана в главной программе, которая содержит начальные и заключительные действия для работы всех блоков компилятора. Именно из этой программы вызывается компилятор.

Теперь мы можем уточнить описание функции **nextch**:

```
nextch ( )
 { if ( positionnow.charnumber == LastInLine )
    {   ListThisLine (  );
        if ( в текущей строке обнаружены
              ошибки)
           ListErrors( );
        ReadNextLine( );
        positionnow.linenumber ++;
        positionnow.charnumber = 1;
    }
        else positionnow.charnumber ++;
  ch = line[positionnow.charnumber];
 }
```

2.2.1. Формирование таблицы ошибок

Будем считать, что сообщения об ошибках выводятся сразу после анализа текущей строки. Таким образом, перед началом трансляции очередной строки программы таблица ошибок пуста.

Для описания функции **error** потребуются следующие переменные:

- таблица ошибок текущей строки:

  ```
  struct
     { struct textposition errorposition;
       unsigned errorcode;
     } ErrList [ERRMAX];
  ```

  Значение константы **ERRMAX** здесь определяет наибольшее количество ошибок в строке, о которых будет получать информацию пользователь; сообщения обо всех других ошибках выводиться не будут;

- количество обнаруженных ошибок в текущей строке: **short ErrInx**;
- флажок **short ErrorOverFlow**, принимающий значение **TRUE**, если количество ошибок, обнаруженных в текущей строке, превышает **ERRMAX**, либо принимающий значение **FALSE** — в противном случае.

Таким образом, описание функции **error** принимает вид:

```
void error ( unsigned errorcode,
             textposition position )
{ if ( ErrInx = ERRMAX )
     ErrorOverflow = TRUE;
  else
     { ++ErrInx;
       Errlist[ ErrInx ].errorcode=errorcode;
       Errlist[ ErrInx ].errorposition.linenumber
                          =position.linenumber;
       Errlist[ ErrInx ].errorposition.charnumber
                          =position.charnumber;
     }
}
```

2.2.2. Печать сообщений об ошибках

Сообщение о каждой ошибке печатается в отдельной строке. Формат этой строки имеет вид

```
**№** ^ошибка код ошибки
```

Порядковый номер ошибки здесь обозначен символом «№», а знак «^» должен появиться в той позиции, где обнаружена ошибка. Кроме того, сообщение об ошибке необходимо представить в текстовом виде. Например:

```
**02**    ^ ошибка код 100
******  использование имени не соответствует описанию
```

В конце листинга желательно также напечатать информацию о количестве обнаруженных ошибок. Например:

```
Компиляция окончена: ошибок — 10 !
```

Пример листинга программы

Работает Pascal-компилятор

```
1  program example ( input, output );
2    const c = 3;
3          b = 56;
4    var  a : 'a' .. 'c';
5           k, i : integer;
6    begin
7       read ( k, i );
8       for a := 'a' to 'c' do
9       case a of
```

```
 10           k: i := i * k;
**01**         ^ ошибка код 100
******  использование имени не соответствует описанию
  11         'b' : i := i + 1;
  12          i : k := k + 2;
**02**         ^ ошибка код 100
******  использование имени не соответствует описанию
 13           b: i := i - k;
**03**         ^ ошибка код 147
******  тип метки не совпадает с типом выбирающего выражения
 14           c: i := ( i + k ) * 2
**04**         ^ ошибка код 147
******  тип метки не совпадает с типом выбирающего выражения
 15       end;
 16       writeln( i, k )
 17   end.

Компиляция окончена: ошибок - 4 !
```

Коротко о главном

1. Модуль ввода-вывода выполняет следующие действия:
   - считывает последовательность литер исходной программы и передает их анализатору;
   - формирует листинг.
2. Анализатор запоминает информацию об ошибках в таблице ошибок. Модуль ввода-вывода использует содержимое этой таблицы для печати сообщений об ошибках.
3. Сообщение об ошибке содержит:
   - порядковый номер ошибки;
   - код ошибки;
   - пояснение в текстовом виде.

Задания

1. Опишите функцию **nextch** — модуль ввода-вывода.

(*Примечание*: из-за отсутствия анализатора, формирующего таблицу ошибок, создайте таблицу ошибок вручную.)

2. Разработайте набор тестов для тестирования модуля ввода-вывода.
3. Выполните тестирование модуля ввода-вывода.

Глава 3

Лексический анализатор

3.1. Взаимодействие лексического анализатора с другими частями компилятора

Лексический анализатор формирует символы исходной программы и строит их внутреннее представление. Кроме того, **сканер** распознает и исключает *комментарии*, которые не нужны для дальнейшей трансляции.

Взаимодействие между сканером и синтаксическим анализатором может осуществляться различными способами. Один способ предполагает, что сканер формирует представление исходной программы в виде последовательности символов до начала работы синтаксического анализатора. При другом способе взаимодействия синтаксический анализатор обращается к сканеру всякий раз, когда потребуется новый символ; в этом случае нет необходимости хранить в памяти всю последовательность символов. Далее мы будем считать, что лексический анализатор реализован по второму способу.

Рассмотрим внутреннее представление символов.

Сканер группирует последовательности литер переменной длины в символы, так как в остальных частях компилятора удобнее работать с кодами фиксированной длины. В частности, синтаксическому анализатору безразлично, какой идентификатор встречается в каждом конкретном случае. При этом сканер заменяет конкретное представление каждого символа некоторым кодом; такие коды символов удобно хранить в виде значений целого типа. Для наглядности и удобства программирования дадим коду каждого символа *имя* и введем константы, мнемоника которых отражает смысл используемых в языке символов. Например:

```
#define  star       21    /* * */
#define  slash      60    /* / */
#define  equal      16    /* = */
#define  comma      20    /* , */
#define  semicolon  14    /* ; */
#define  colon       5    /* : */
```

```
#define  point          61    /* . */
#define  arrow          62    /* ^ */
#define  leftpar         9    /* ( */
#define  rightpar        4    /* ) */
#define  lbracket       11    /* [ */
#define  rbracket       12    /* ] */
#define  flpar          63    /* { */
#define  frpar          64    /* } */
#define  later          65    /* < */
#define  greater        66    /* > */
#define  laterequal     67    /* <= */
#define  greaterequal   68    /* => */
#define  latergreater   69    /* <> */
#define  plus           70    /* + */
#define  minus          71    /* - */
#define  lcomment       72    /* (* */
#define  rcomment       73    /* *) */
#define  assign         51    /* := */
#define  twopoints      74    /* .. */
#define  ident           2    /* идентификатор */
#define  floatc         82    /* вещественная
                                  константа */
#define  intc           15    /* целая константа */
#define  charc          83    /* символьная
                                  константа */
```

В дальнейшем мы будем считать, что имена констант, соответствующих кодам служебных слов, состоят из имени служебного слова и окончания «**sy**»:

```
#define casesy      31
#define elsesy      32
#define filesy      57
#define gotosy      33
#define thensy      52
#define typesy      34
#define untilsy     53
#define dosy        54
#define withsy      37
#define ifsy        56
```

и т. д.

При обнаружении ошибки на этапе анализа программы необходимо напечатать соответствующее сообщение и указать место, где обнаружена ошибка. Поэтому сканер должен формировать *координаты каждого символа*, а именно: номер строки исходной программы и позицию символа (позицию первой литеры символа) в этой строке:

```
unsigned symbol;                /* код символа */
struct textposition token;      /* позиция символа */
```

Для семантического анализа необходимо знать представление идентификаторов в виде последовательности букв и/или цифр, а для генерации кода — значения констант. Поэтому сканер заносит имена констант, типов, переменных, процедур и функций в **таблицу имен**, а значение констант — в одну из следующих переменных (в зависимости от типа константы):

```
int    nmb_int;     /* целая константа */
float  nmb_float;   /* вещественная константа */
char   one_symbol;  /* символьная константа */
```

Поиск может быть выполнен более эффективно, если элементы таблицы имен *упорядочены по некоторому признаку*. В нашем случае, когда элементами таблицы являются строки литер, наиболее подходящим является упорядочивание с использованием *хеширования*.

В процессе лексического анализа каждый идентификатор заносится в таблицу имен только один раз, но поиск по этой таблице ведется всегда, когда встречается этот идентификатор. Для поиска и занесения идентификатора в таблицу имен воспользуемся функцией:

```
search_in_table (char *name)
```

Результаты этой функции — значения внешних переменных:

```
unsigned hashresult; /* hash-функция
                        для заданного имени */
char *addrname;      /* адрес имени в
                                таблице имен */
```

Таким образом, для хранения информации о каждом символе требуются следующие внешние переменные:

```
unsigned symbol;          /* код символа */
struct textposition token; /* позиция символа */
char *addrname;           /* адрес идентификатора
                             в таблице имен */
int nmb_int;              /* значение целой
                             константы */
float nmb_float;          /* значение вещественной
                             константы */
char one_symbol;          /* значение символьной
                             константы */
```

3.2. Программирование лексического анализатора

Рассмотрим структуру функции **nextsym**, которая вызывается всякий раз, когда синтаксический анализатор запрашивает очередную лексему.

В исходной программе смежные символы могут быть разделены произвольным количеством пробелов, но внутри символов пробелы не допускаются. Учитывая также, что в переменной **ch** находится значение очередной сканируемой литеры, запишем первоначальную версию **nextsym**:

```
nextsym ( )
  { while ( ch == ' ' ) nextch ( );
    token.linenumber = positionnow.linenumber;
    token.charnumber = positionnow.charnumber;
    сканировать символ;
  }
```

Действие «*сканировать символ*» определяется правилами, задающими синтаксис символов. Запишем эти правила следующим образом:

```
<символ> ::=
   <идентификатор или ключевое слово>|
   <целая константа> | <вещественная константа>|
   <символьная константа>|
   <= | < > |
   >= | > |
   := | : |
```

```
+ |
- |
* | *) |
/ |
= |
( | (* |
) |
{ |
} |
' |
[ |
] |
. | .. |
, |
^ |
;
```

Отсюда видно, что:

1) альтернативы каждой строки отличаются от альтернатив всех остальных строк по первой литере;
2) одна и та же литера, в зависимости от контекста, может иметь различный смысл. Например, двоеточие представляет собой отдельный символ, если появляется при описании переменных:

```
a , b : integer;
```

но эта же литера в сочетании со знаком «=» образует символ присваивания. Поэтому при появлении во входном потоке двоеточия нужно обязательно просмотреть следующую литеру.

Таким образом, зная значение первой литеры сканируемого символа, нужно выбрать один из возможных способов дальнейшего анализа. В этом случае удобно воспользоваться конструкцией «выбор»:

```
switch (ch)
   { case  <буква> :
        сканировать идентификатор
                    или ключевое слово;
       break;
     case <цифра> :
       сканировать целую или
                  вещественную константу;
       break;
     case '/'' :
       сканировать символьную константу;
       break;
```

```
      case ':' :
         сканировать := или :;
         break;
      case  '<' :
         сканировать < > или <= или < ;
         break;
      case  '>' :
         сканировать >= или >;
         break;
      case  '.' :
         сканировать . или .. ;
         break;
           . . .
   }
```

После выхода из функции **nextsym** переменная **ch** всегда содержит литеру, следующую за распознанным символом. В ряде случаев только по этой литере можно завершить формирование символа (например, при анализе «<», «>», «:» и др.). В качестве примера рассмотрим сканирование символов «<», «<=», «<>», «:», «:=» и «;»:

```
case '<': nextch( );
         if ( ch  ==  '=' )
           {
            symbol = laterequal ;  nextch( );
           }
          else
            if ( ch  ==  '>' )
              {
               symbol = latergreater;  nextch( );
              }
             else
               symbol = later;
          break;
case ':' : nextch( );
           if ( ch  ==  '=')
             {
              symbol  =  assign; nextch( );
             }
            else
              symbol  =  colon;
           break;
case ';': symbol = semicolon;
           nextch( );
           break;
```

При сканировании целой или вещественной константы необходимо учитывать, что в вычислительной машине могут быть представлены числа лишь из определенного диапазона. Например, пусть архитектура компьютера такова, что для хранения целого числа используется 16 разрядов. Максимальное положительное число, которое можно записать в 16-разрядную ячейку, равно 32767. Использование же констант, которые превосходят это значение, должно приводить к сообщению об ошибке с кодом 203 — «целая константа превышает предел».

Воспользуемся константой **maxint** для хранения максимального положительного числа и опишем лексический анализ целой константы:

```
case <цифра> :
      nmb_int = 0;
      while( ch >= '0' && ch <= '9' )
        {
          digit=ch-'0';
          if  ( nmb_int < maxint /10 ||
                ( nmb_int == maxint/10 &&
                      digit <= maxint % 10 ))
               nmb_int = 10 * nmb_int + digit;
           else
             {
              /* константа превышает предел */
              error (203, positionnow );
              nmb_int = 0;
             }
         nextch();
        }
      symbol = intc;
      break;
```

Теперь рассмотрим сканирование идентификатора. Для хранения идентификатора воспользуемся символьным массивом:

```
char  name  [MAX_IDENT];
```

а в переменной

```
unsigned  lname;
```

будем хранить длину идентификатора:

```
case <буква> :
    lname = 0;
    while ((( ch>= 'a'  && ch <= 'z') ||
            ( ch >= 'A' && ch <= 'Z' ) ||
            (ch >= '0'  && ch <= '9' )) &&
                          lname < MAX_IDENT)
              { name[ lname++ ] = ch;
                nextch( );
              }
    name [ lname ] = '\0';
    symbol = код идентификатора или
                        код ключевого слова
    break;
```

Чтобы выяснить, является ли текущий идентификатор ключевым словом, или же это идентификатор, введенный пользователем, лексическому анализатору необходима **таблица ключевых слов**, которая содержит все ключевые слова языка программирования и соответствующие им коды. Сканер осуществляет поиск по этой таблице всякий раз, когда в исходной программе встречается идентификатор. Так как на этот процесс тратится много времени, важно выбрать такую организацию таблицы, которая допускала бы эффективный поиск.

Рассмотрим один из возможных способов. Разделим все ключевые слова на несколько групп, где признак группы — это длина слова (количество литер в слове). Тогда для стандарта языка Паскаль имеет место следующее распределение ключевых слов по группам:

- группа 1 — нет;
- группа 2 — **if**, **do**, **of**, **or**, **in**, **to**;
- группа 3 — **end**, **var**, **div**, **and**, **not**, **for**, **mod**, **nil**, **set**;
- группа 4 — **then**, **else**, **case**, **file**, **goto**, **type**, **with**;
- группа 5 — **begin**, **while**, **array**, **const**, **label**, **until**;
- группа 6 — **downto**, **packed**, **record**, **repeat**;
- группа 7 — **program**;
- группа 8 — **function**;
- группа 9 — **procedure**.

Все эти ключевые слова расположим в таблице по возрастанию номеров групп, а после каждой группы слов добавим до-

полнительную строку, содержащую код идентификатора, введенного пользователем:

```
struct key
  { unsigned codekey;
    char namekey [9];
  } keywords [  ] =
    {
      { ident, " " },
      { dosy, "do" },
      { ifsy, "if" },
      { insy, "in" },
      { ofsy, "of" },
      { orsy, "or" },
      { tosy, "to" },
      { ident, " " },
      { andsy, "and" },
      { divsy, "div" },
        . . .
      { proceduresy, "procedure" },
      { ident, " " }
    };
```

Введем вспомогательный массив:

```
unsigned short last [ ] =
       { -1, 0, 7, 17, 25, 32, 37, 39, 41, 43 }
```

где значение элемента **last [i]** — это номер строки таблицы ключевых слов с кодом **ident** для группы **i**. В этом случае поиск по таблице сводится к следующему:

```
{ strcpy (keywords[last[lname]].namekey, name );
  i = last[lname-1] + 1;
  while (strcmp(keywords[i].namekey,name) != 0 )
         i ++;
  symbol = keywords[i].codekey;
}
```

Лексический анализ остальных символов программируется достаточно просто.

3.2.1. Лексические ошибки

В процессе формирования символов сканер может встретить литеру, которая не используется в языке Паскаль, — например, «?», «&» или «%». В этом случае формируется сообщение об ошибке — «запрещенный символ». Если вместо символьной константы 'a' сканер обнаружит '', то в листинге появится сообщение «ошибка в символьной константе». Кроме того, на этапе лексического анализа проверяются значения числовых констант.

Пример листинга программы с сообщениями о лексических ошибках:

```
Работает Pascal-компилятор
   1  program example ( output );
   2   var i, j : real;
   3       mas : array [ 1 .. 10 ]  of  char;
   4   begin  i := 1;  j := 32769;
**01**                          ^ ошибка код 203
****** константа превышает предел
   5      ?             $      &
**02**    ^ ошибка код 6
****** запрещённый символ
**03**                  ^ ошибка код 6
****** запрещённый символ
**04**                          ^ ошибка код 6
****** запрещённый символ
   6    i := i + ( i*j ) + 100; writeln( i );
   7    for i := 1 to 10
   8      if odd ( i )
   9       then a [ i ] := 'a'
  10        else  a [ i ] := '' ;
 **05**                       ^ ошибка код 6
 ****** ошибка в символьной константе
  11      writeln( a[ i ] );
  12   end.

Компиляция окончена: ошибок - 5 !
```

Коротко о главном

1. Лексический анализатор (сканер) формирует символы исходной программы и исключает комментарии.
2. Коды символов удобно хранить в виде значений целого типа.
3. Лексический анализатор должен передавать другим блокам компилятора следующую информацию: код символа, позицию символа, значение константы, адрес идентификатора в таблице имен.
4. По значению первой литеры сканируемого символа выбирается способ анализа символа.
5. Сканер формирует сообщения об ошибках, если обнаруживает слишком большую константу, не используемую в языке литеру или ошибку в символьной константе.

Задания

1. Опишите функцию **`nextsym`** — лексический анализатор для стандарта языка Паскаль.
2. Разработайте набор тестов для тестирования лексического анализатора.
3. Выполните тестирование лексического анализатора.

Глава 4

Синтаксический анализатор

Опишем структуру **синтаксического анализатора**, а затем дополним его **средствами семантического анализа.**

В предыдущей главе мы неформально проанализировали правила, описывающие синтаксис символов, и составили соответствующие программные фрагменты сканера. Аналогичный подход может быть использован и при написании синтаксического анализатора.

Все правила, описывающие синтаксис языка, имеют вид:

```
<S> ::= L
```
(4.1)

где **S** — обозначение синтаксической конструкции, а **L** может состоять из символов языка, синтаксических конструкций и метасимволов «|» и «{ }».

Основная идея написания синтаксического анализатора заключается в следующем.

Для каждого правила (4.1) необходимо описать функцию, тело которой является результатом некоторого преобразования правой части этого правила:

```
procs ( )
 { T( L )
 }
```

При описании функции, соответствующей синтаксической конструкции **S**, мы будем придерживаться следующих соглашений:

а) перед обращением к функции переменная **symbol** содержит код первого символа, который должен быть проанализирован этой функцией;

б) функция проверяет, выводима ли входная последовательность символов из **S**, и в случае ошибки формирует соответствующее сообщение;

в) перед тем как выйти из функции, в переменной **symbol** запоминается код символа, который встретился сразу после анализа конструкции **S**.

Преобразование **T(L)** определяется следующим набором правил:

- **правило 1:** если **L** состоит из единственного символа, то **T(L)--> accept(L)** (где **-->** имеет смысл глагола «есть»). При этом функция **accept** имеет следующую структуру:

```
if (сканируемый символ совпадает с ожидаемым)
    сканировать следующий символ;
  else   сформировать сообщение об ошибке;
```

Уточненный вариант этой функции:

```
void accept(unsigned symbolexpected
                      /* код ожидаемого символа */)
 { if (symbol == symbolexpected)
       nextsym();
    else  error (symbolexpected, token);
 }
```

Примечание: коды символов должны совпадать с номерами сообщений об ошибках вида «должен идти символ ... ». Например, код символа «(» равен 9, поэтому сообщение об ошибке «должен идти символ (» также имеет номер 9;

- **правило 2:** если **L** состоит из единственной синтаксической конструкции (например, **<A>**), то **T(L)--> A**, где **A** — имя функции для правила **<A>::=Q** ;
- **правило 3:** если **L** — последовательность вида: **L1 L2 L3...Ln**, где **Li(i=1,n)** — символ языка или синтаксическая конструкция, то:

```
T (L1 L2 ... Ln) --> T (L1)
                     T (L2)
                     ...
                     T (Ln)
```

Здесь преобразование **T(Li)** выполняется по *правилу 1*, если **Li** — символ языка, и по *правилу 2*, если **Li** — синтаксическая конструкция.

Пример 4.1. Функции синтаксического анализа для различных программных конструкций:

1) <*программа*>::=**program**<*имя*>;<*блок*>. (упрощенный вариант правила для конструкции <*программа*> по сравнению со стандартом языка Паскаль)

```
void programme ( )
   /* анализ конструкции <программа> */
   {  accept (programsy);
      accept (ident);
      accept (semicolon);
      block ( ); /* анализ конструкции <блок> */
      accept (point);
   }
```

2) *<блок>*::=*<раздел меток><раздел констант>*
<раздел типов><раздел переменных>
<раздел процедур и функций><раздел операторов>

```
void block( )
   /* анализ конструкции <блок> */
   { labelpart ( );  constpart ( );
     typepart ( );   varpart ( );
     procfuncpart ( );  statementpart ( );
   }
```

3) *<цикл с предусловием>*::=**while***<выражение>***do**
<оператор>

```
void whilestatement ( )
   /* анализ конструкции <цикл с предусловием> */
   { accept (whilesy); expression( );
     accept (dosy);   statement ( );
   }
```

4) *<цикл с параметром>*::=**for***<параметр цикла>*:=
*<выражение><направление><выражение>***do***<оператор>*
<параметр цикла>:=*<имя>*
<направление>::=**to**|**downto**

```
void forstatement ( )
  /* анализ конструкции <цикл с параметром> */
  { accept (forsy);  accept (ident);
    accept (assign);  expression ( );
    if (symbol==tosy || symbol==downtosy)
        nextsym( );
    expression( );  accept (dosy);
    statement( );
  }
```

- **правило 4:** если **L** содержит несколько альтернатив, т. е. имеет вид **Y1|Y2|...|Ym**, то необходимо обеспечить анализ подходящей альтернативы.

Введем обозначение: **start(Yi)** — символы, с которых могут начинаться последовательности символов, получаемые из **Yi(i=1,n)**. Например, конструкция *<выражение>* может начинаться с одного из следующих символов: «+», «−», «(», «[», «**not**», «*<имя>*», «*<константа>*», «**nil**». Следовательно, **start (***<выражение>***)={plus,minus,leftpar,lbracket,notsy,ident, intc,floatc,charc,nilsy}**. Тогда:

```
T (Y1|Y2| … |Ym) -->
  switch (symbol)
    {  case start(Y1) : T(Y1);  break;
       case start(Y2) : T(Y2);  break;
        ...
       case start(Ym) : T(Ym );
    };
```

Выбор **T(Yi)** должен однозначно определяться на основании очередного сканируемого символа, ведь в дальнейшем нам предстоит выполнять семантический анализ параллельно с синтаксическим. И если **T(Yi)** выбирается неоднозначно, то может произойти возврат, и придется уничтожить результаты семантической обработки, что в большинстве случаев сделать достаточно сложно.

Требование 1

Если синтаксическое правило имеет вид:

<S> ::=Y1|Y2|...|Ym

то множества **start(Yi) (i=1,m)** попарно не пересекаются, т. е.

start(Yi)*start(Yj)=<пусто>(i<>j)

где ***** — обозначение операции «пересечение множеств».

Следует также обратить особое внимание на те правила, у которых одна из альтернатив — **<пусто>**.

Требование 2

Если синтаксическое правило имеет вид

<S>::=Y1|Y2|...|Ym| <пусто>

то

start(Yi)*follow(S)=<пусто>(i=1,m)

где **follow (S)** — множество символов, которые могут встретиться сразу после анализа конструкции **S**. Например,

follow (<раздел переменных>**)= {procsy,funcsy,beginsy}**

Если правило содержит в правой части только две альтернативы, то удобно воспользоваться оператором **if(...)...; else...;**, а если одна из двух альтернатив — **<пусто>**, то неполным условным;

- **правило 5**: если **L** имеет вид **{Z}**, то:

```
T(L)--> while(symbol принадлежит start (Z)) T(Z);
```

Если **Z** встречается в правой части правила хотя бы один раз (**L** есть **Z{Z}**), то:

```
T(L)--> T(Z);
        while(symbol принадлежит start(Z)) T(Z);
```

или

```
T(L)--> do T(Z);
         while (symbol принадлежит start (Z));
```

Пример 4.2. Функции синтаксического анализа конструкций:

1) <составной оператор>::=**begin**<оператор> {;<оператор>}**end**

```
void compoundstatement ( )
  /* анализ конструкции <составной оператор> */
   { accept (beginsy); statement( );
     while (symbol == semicolon)
          { nextsym( ); statement ( );
          }
      accept (endsy);
  }
```

2) <раздел переменных>:= **var** <описание однотипных переменных>;{<описание однотипных переменных>;} | <пусто>

Прежде чем описать анализ конструкции <раздел переменных>, проверим, не нарушено ли *требование 2* (одна из альтернатив — **<пусто>**). Символы, которые могут следовать после анализа раздела переменных, — **procsy**, **funcsy**, **beginsy**. Следовательно, *требование 2* выполнено.

```
void  varpart( )
   /* анализ конструкции <раздел переменных> */
     { if (symbol == varsy)
         { accept (varsy);
           do
              { vardeclaration ( );
                accept (semicolon);
              }
           while (symbol == ident);
         }
   }
```

Примечание: вызов **accept(varsy)** можно заменить на обращение к функции **nextsym**, так как в рассматриваемом контексте сканируемый символ совпадает с ожидаемым.

3) <описание однотипных переменных>::=<*имя*>{,<*имя*>}:
<*тип*>

```
void  vardeclaration ( )
  /* анализ конструкции
     <описание однотипных переменных> */
    {  accept ( ident );
       while (symbol == comma)
             { nextsym ( );
               accept (ident);
             }
       accept (colon);
       type ( );
    }
```

4) <*регулярный тип*>::= **array**[<*простой тип*>
{,<*простой тип*>}] **of** <*тип компоненты*>
<*тип компоненты*> ::= <*тип*>

```
void  arraytype()
  /* анализ конструкции <регулярный тип> */
  { accept (arraysy); accept (lbracket);
    simpletype( );
    while (symbol == comma)
         {
           nextsym( );  simpletype( );
         }
    accept (rbracket);  accept (ofsy); type ( );
  }
```

5) *<переменная>*::=*<имя>*{[*<выражение>*{,*<выражение>*}]|.*<имя>*|^}

```
void variable ( )
  /* анализ конструкции  <переменная> */
  { accept (ident);
    while (symbol == lbracket || symbol == point
               || symbol == arrow)
       switch (symbol)
          { case lbracket :
                nextsym ( ); expression ( );
                while(symbol == comma)
                  {
                    nextsym( ); expression( );
                  }
                accept (rbracket);
                break;
            case point :
                nextsym ( ); accept (ident);
                break;
            case arrow :
                nextsym ( );
                break;
          }
  }
```

Особенности анализа конструкций языка

Некоторые сложности возникают при анализе конструкций *<простой тип>*, *<оператор>*, *<множитель>* и др.:

1) *<простой тип>*::=*<перечислимый тип>* | *<ограниченный тип>* | *<имя типа>*
2) *<перечислимый тип>*::=(*<имя>*{,*<имя>*})
3) *<ограниченный тип>*::=*<константа>*..*<константа>*
4) *<имя типа>*::=*<имя>*
5) *<константа>*::=*<имя константы>* |...
6) *<имя константы>*::=*<имя>*
7) *<оператор>*::=*<оператор присваивания>* | *<оператор процедуры>* | ...
8) *<множитель>* ::= *<переменная>* | *<вызов функции>* | *<константа>* | ...

В этом случае нарушено *требование 1* (альтернативы правой части *правил 1, 7* и *8* начинаются с символа *<имя>*). Так как синтаксический и семантический анализ выполняется параллельно, будем предполагать, что семантическая проверка поможет синтаксическому анализатору осуществить выбор подходящей альтернативы.

Теперь рассмотрим анализ условного оператора:

<условный оператор>::= **if** *<выражение>* **then** *<оператор>* |
if *<выражение>* **then** *<оператор>* **else** *<оператор>*

Требование 1 здесь нарушено (обе альтернативы начинаются с символа **if**). Преобразуем это правило в:

<условный оператор> ::= **if** *<выражение>* **then** *<оператор>*
<хвост><*хвост*> ::= **else** *<оператор>* | *<пусто>*

Теперь выполнено *требование 1*, однако для вложенных условных операторов: **if ... then ... if ... then ... else ...** нарушается *требование 2*, так как *<хвост>* может начинаться с символа **else**, а за конструкцией *<хвост>* может следовать символ **else** вложенного условного оператора. Поэтому примем следующее соглашение: каждый **else** соответствует ближайшему к нему слева условию. Тогда описание функции синтаксического анализа для конструкции *<условный оператор>* будет иметь вид:

```
void ifstatement( )
/* анализ конструкции <условный оператор> */
   {  accept (ifsy);  expression ( );
      accept (thensy);  statement ( );
      if (symbol == elsesy)
         { nextsym ( );
           statement ( );
         }
   }
```

Правила 1–5 позволяют достаточно быстро написать функции синтаксического анализа, используя формальные правила описания синтаксиса языка программирования (формы Бэкуса—Наура или диаграммы Вирта). Работа анализатора на-

чинается с вызова функции, соответствующей синтаксическому правилу *<программа>*. Эта функция обращается к другим функциям, которые в свою очередь снова вызывают функции, и т. д. Для анализа некоторых конструкций (например, *<оператор>* и *<выражение>*) могут потребоваться рекурсивные вызовы функций.

Таким образом, синтаксический анализатор строится по принципу **детерминированного рекурсивного нисходящего анализатора**.

Анализатор, созданный по *правилам 1–5*, правильно анализирует программы, которые удовлетворяют формальным правилам описания языка. Фактически такой анализатор работает верно только до первой ошибки в исходной программе. Чтобы продолжить анализ после обнаружения ошибки, необходимо усовершенствовать алгоритм работы анализатора.

Пример листинга программы, полученного в результате работы синтаксического анализатора, написанного по правилам 1–5:

Работает Pascal-компилятор

```
 1  program Evklid;
 2  { определение наибольшего общего делителя }
 3    var M, N :integer;
 4    begin
 5       writeln (' Введите M и N ' );
 6       readln ( M, N );
 7       while  M <> N  do
 8          begin
 9            if M > N
10               then M := M-N
11               else N := N - M
12          end;
13       write ('НОД=', M )
14    end.

Компиляция окончена: ошибок нет!
```

Коротко о главном

1. Все правила, описывающие синтаксис языка, имеют вид: **<S> ::= L**.
2. Чтобы создать синтаксический анализатор, для каждого правила языка **<S> ::= L** нужно описать функцию, тело которой является результатом преобразования правой части этого правила: **procs( ){T(L)}**.
3. Преобразование **T(L)** определяется пятью правилами написания синтаксического анализатора. Использование этих правил накладывает определенные требования на описание синтаксиса языка программирования.
4. Синтаксический анализатор Паскаль-программ строится по принципу детерминированного рекурсивного нисходящего анализатора.

Задания

1. Опишите функции синтаксического анализа для конструкций стандарта языка Паскаль.
2. Разработайте набор тестов для тестирования синтаксического анализатора.
3. Выполните тестирование синтаксического анализатора.

Примечание: синтаксический анализатор, написанный по правилам 1–5, должен правильно транслировать синтаксически правильные Паскаль-программы (с точки зрения формальных правил описания языка).

Глава 5

Нейтрализация синтаксических ошибок

Синтаксический анализатор, рассмотренный в главе 4, проверяет, удовлетворяет ли исходная программа формальным правилам описания языка, и определяет синтаксическую структуру этой программы. Однако этот анализатор правильно функционирует только до обнаружения первой синтаксической ошибки. Если входная последовательность символов содержит неправильную конструкцию, то анализ программы прекращается. Очевидно, что на практике такая организация работы неприемлема: вместо этого компилятор должен сформировать соответствующее сообщение и продолжить анализ, так как за один запуск программы необходимо обнаружить наибольшее количество ошибок.

Дополним анализатор действиями, позволяющими продолжить анализ исходной программы после обнаружения ошибки. Эти действия называются **нейтрализацией ошибки.**

Основная идея нейтрализации ошибок, которую мы будем рассматривать, состоит в следующем: после выявления ошибки надо пропустить один или несколько символов, чтобы найти символ, начиная с которого можно возобновить анализ. Опишем этот процесс более подробно.

Пусть функция с именем **procs** соответствует синтаксическому правилу **<S> ::= L**. Множество символов, с которых может начинаться конструкция **S**, обозначим как **starters**. В начало функции **procs** добавим проверку, является ли текущий символ допустимым начальным символом анализируемой конструкции:

```
if (!belong (symbol, starters))
      {   error ( ... );                          (5.1)
          skipto (starters);
      }
```

Здесь функция **belong** проверяет принадлежность символа **symbol** множеству **starters**. Функция **skipto** пропускает символы входного потока, пока не встретится один из символов множества, указанного в качестве параметра. Как только будет найден искомый символ, начинается анализ конструкции **S**.

Пример. Пусть последовательность литер:

```
* 5 * * 2 * A, B, C : integer;
```

передается для анализа функции **vardeclaration** (см. главу 4). Здесь **starters = [ident]**, а текущий сканируемый символ — «*». В соответствии с принятым соглашением, функция **vardeclaration** сформирует сообщение об ошибке, пропустит литеры «*** 5 * * 2 ***» и начнет анализ описания однотипных переменных.

Однако из-за ошибки в исходной программе начальный символ конструкции может вообще не встретиться. Тогда **skipto** пропустит символы, которые должны быть обработаны функцией, вызвавшей **procs**. Чтобы избежать этой ситуации, **skipto** должна просматривать входной текст, пока не будет найден один из символов, который по правилам описания языка может следовать за конструкцией **S**. Назовем эти символы *внешними*. Множество внешних символов передается в точку вызова функции **procs** через параметр.

С учетом сделанных замечаний, добавим формальный параметр в заголовок функции: **procs(followers)** и перепишем (5.1) в несколько иной форме:

```
if (!belong(symbol, starters))
      {
        error(...);
        skipto2 (starters, followers);
      }
if (belong (symbol, starters))
     /*анализ конструкции S */
      T (L);
```

Здесь функция **skipto2** осуществляет проверку на принадлежность символа одному из множеств, указанных в качестве фактических параметров.

Появление внешнего символа в результате обращения к функции **skipto2** свидетельствует об ошибке в исходной программе, но этот символ ни в коем случае нельзя пропускать!

Через параметр **followers** вызывающая функция сообщает, начиная с каких символов она продолжит анализ после окончания работы **procs**. Поэтому проверку синтаксической правильности конструкции мы завершим следующим образом:

```
if (!belong(symbol, followers))
    {
       error(...)
       skipto(followers);
    }
```

Тогда в итоге структура функции **procs** примет вид:

```
procs (followers)
      {
        if (!belong(symbol, starters))
              {
                error(...);
                skipto2 (starters, followers);
              }
        if (belong (symbol, starters))
             { /*анализ конструкции*/
                T(L);
                if (!belong(symbol, followers))
                     {
                       error(...);
                       skipto(followers);
                     }
              }
      }
```

Теперь рассмотрим, как формируется значение фактического параметра при обращении к **procs**. Ясно, что это значение включает в себя множество символов, которые анализатор ожидает ***сразу*** после анализа **S**. Однако в некорректной программе эти символы могут быть пропущены. Следовательно, к значению фактического параметра необходимо добавить множество символов, которые получила вызывающая функция в качестве параметра.

Пример 5.1. Конструкции языка, соответствующие им функции и множества их внешних символов:

Конструкция	Имя функции	Множество внешних символов
<программа>	**programme** ↓	
<блок>	**block** ↓	.
<раздел переменных>	**varpart** ↓	procedure function begin .
<описание однотипных переменных>	**vardeclaration** ↓	; procedure function begin .
<тип>	**type**	; procedure function begin .

Примечание: подчеркнуты символы, которые могут следовать ***сразу*** после анализа конструкции.

Таким образом, если функция синтаксического анализа обнаруживает ошибку, то она не сообщает о случившемся вызвавшей ее функции, а продолжает просмотр входного текста до того места, откуда можно возобновить анализ. Внешние символы, передаваемые в качестве параметров, называют *символами возобновления*.

Компилятор, обеспечивающий выдачу сообщений об ошибках и нейтрализацию ошибок, имеет, как следствие, несколько больший объем, но это вполне компенсируется повышением производительности труда пользователя.

Для хранения множеств символов **starters** и **followers** удобно воспользоваться *битовыми строками*. Так как коду каждого символа поставлено в соответствие целое число (см. главу 3), то состояние *i*-го разряда в битовой строке отражает наличие или отсутствие символа с кодом *i* в множестве. Тогда описание функции **belong** можно представить так:

```
boolean belong (
   unsigned element /* номер искомого элемента */,
   unsigned *set    /* множество, в котором
                                    ищем элемент */)
/* поиск элемента element в множестве set;
 функция  возвращает значение "истина", если
 элемент  присутствует в множестве, иначе — "ложь" */
  {
    unsigned  word_number /* номер слова, в котором
                     находится информация о символе
                                  с кодом element */,
    bit_number /* номер бита, в котором находится
              информация о символе с кодом element */;
    word_number = element / WORDLENGTH ;
  /* WORDLENGTH — длина минимально адресуемой
    единицы памяти в битах */
    bit_number = element % WORDLENGTH;
    return(set [word_number] &
            (1 << (WORDLENGTH-1 - bit_number)));
  }
```

Пример 5.2. Функции синтаксического анализа с учетом нейтрализации синтаксических ошибок для конструкций *<блок>*, *<раздел переменных>* и *<описание однотипных переменных>*.

Воспользуемся функцией:

```
void set_disjunct ( unsigned set1 [ ],
                    unsigned set2 [ ],
                    unsigned set3 [ ] )
```

результатом работы которой является битовая строка **set3** — объединение битовых строк **set1** и **set2**.

Введем внешние переменные:

```
unsigned
*begpart          /* указатель на битовую строку
                     для множества символов label,
                     const, type, var, function,
                     procedure, begin */,
*st_typepart      /* указатель на битовую строку
                     для множества символов type,
                     var,  function, procedure,
                     begin */,
*st_varpart       /* указатель на битовую строку
                     для множества символов var,
                     function, procedure, begin */,
*st_procfuncpart  /* указатель на битовую строку
                     для множества символов
                     function, procedure, begin */,
*id_starters      /* указатель на битовую строку
                     для символа <имя> */,
*after_var        /* указатель на битовую строку
                     для символа ";" — символа,
                     ожидаемого сразу после анализа
                     конструкции <описание
                     однотипных переменных> */;
```

Значения битовых строк и указатели на них должны быть проинициализированы в главной программе компилятора.

Примечание: так как позиция текущего символа содержится во внешней переменной **token**, в дальнейшем мы будем пользоваться вариантом функции **error** с одним параметром — кодом ошибки.

В результате функции синтаксического анализа с учетом нейтрализации синтаксических ошибок примут вид:

```
void block (unsigned *followers)
  {
    unsigned ptra [SET_SIZE] /* битовая строка
                 для хранения внешних символов */;
    if (!belong (symbol, begpart))
      {
        error (18); /* ошибка в разделе описаний */
        skipto2 (begpart, followers);
      }
    if (belong(symbol, begpart))
      {
        labelpart ( );
        set_disjunct (st_typepart, followers,ptra);
        constpart (ptra);
        set_disjunct (st_varpart, followers, ptra);
        typepart (ptra);
        set_disjunct ( st_procfuncpart,
                                   followers, ptra);
        varpart (ptra);
        procfuncpart (ptra);
        statpart (followers);
        if (!belong symbol, followers))
          {
            error (6); /*  запрещенный символ */
            skipto (followers);
          }
      }
  }


void varpart (unsigned *followers)
  {
    unsigned ptra [ SET_SIZE ] /* битовая строка
                    для хранения внешних символов */;
    if (!belong (symbol, st_varpart)
      {
        error (18);  /* ошибка в разделе описаний */
        skipto2 ( st_varpart, followers );
      }
```

```
    if (symbol == varsy)
        {
          nextsym ( );
          set_disjunct (after_var, followers, ptra);
          do
             {
                vardeclaration (ptra);
                accept (semicolon);
             }
          while (symbol == ident);
          if (!belong (symbol, followers))
            {
               error (6); /* запрещенный символ */
               skipto (followers);
            }
        }
  }

void vardeclaration (unsigned *followers)
  {
    if (!belong (symbol, idstarters))
      {
         error (2); /* должно идти имя */
         skipto2 (idstarters, followers);
      }
    if (symbol == ident)
      {
        nextsym ( );
        while (symbol == comma)
          {
             nextsym( );
             accept (ident);
          }
        accept (colon);
        type (followers);
        if (!belong(symbol, followers))
          {
            error (6); /* запрещенный символ */
            skipto (followers);
          }
      }
  }
```

Пример листинга программы с сообщениями об обнаруженных синтаксических ошибках в разделе объявления переменных (с точки зрения формальных правил):

Работает Pascal-компилятор

```
   1  program example( a, c, v );
   2  var ***1** a, l :integer;
 **01**   ^ ошибка код  2
 ****** должно идти имя
   3      b,f (a , , f);
 **02**        ^ ошибка код    5
 ****** должен идти символ  ':'
 **03**             ^ ошибка код    2
 ****** должно идти имя
   4     f1, f1:(a1,a2, ,a3) 56;
 **04**                    ^ ошибка код    2
 ****** должно идти имя
 **05**                          ^ ошибка код    6
 ****** запрещенный символ
   5     mas1,mas2 :
   6      array [(s,d),'d'..d,] integer;
 **06**                           ^ ошибка код 10
 ****** ошибка в типе
 **07**                              ^ ошибка код  8
 ****** должно идти  OF
   7     f,k :2..6;
   8     d, : real;
 **08**      ^ ошибка код    2
 ****** должно идти имя
   9     d1,,d2: ;  ***
 **09**      ^ ошибка код    2
 ****** должно идти имя
 **10**           ^ ошибка код   10
 ****** ошибка в типе
 **11**               ^ ошибка код    6
 ****** запрещенный символ
  10
  11  begin
  12
  13  end.

Компиляция окончена: ошибок - 11 !
```

Коротко о главном

1. Действия, позволяющие продолжить анализ программы после обнаружения ошибки, называются нейтрализацией ошибок.
2. Основная идея нейтрализации синтаксических ошибок — после выявления ошибки надо пропустить один или несколько символов, чтобы найти символ, начиная с которого можно возобновить анализ.
3. Чтобы реализовать нейтрализацию синтаксических ошибок, необходимо:
   - сформировать множество внешних символов для каждой функции анализа;
   - включить в начало и в конец каждой функции дополнительные действия.

Задания

1. Создайте битовые строки для множеств символов, с которых могут начинаться различные конструкции стандарта языка Паскаль.
2. Создайте битовые строки для множеств символов, которые могут следовать сразу после различных конструкций стандарта языка Паскаль.
3. Дополните синтаксический анализатор действиями по нейтрализации синтаксических ошибок.
4. Разработайте набор тестов для нейтрализации синтаксических ошибок.
5. Выполните тестирование синтаксического анализатора, содержащего действия по нейтрализации синтаксических ошибок.

Глава 6

Семантический анализатор

6.1. Контекстные условия

Формальные правила описания синтаксиса языка программирования (формы Бэкуса—Наура, а также синтаксические диаграммы) служат основой для построения синтаксического анализатора, однако они дают неполное определение языка. Язык определяется с помощью формальных и неформальных описаний. Синтаксические правила языка программирования, которые задаются с помощью естественного языка (неформально), называются **контекстными условиями**.

Наличие контекстных условий в языке Паскаль связано с его следующими особенностями:

1) идентификаторы используются для именования различных конструкций — типов, констант, переменных, процедур, функций. Как правило, по виду идентификатора нельзя определить способ его использования;
2) в программах один и тот же идентификатор может использоваться для обозначения различных типов. Поэтому при проверке синтаксической правильности программы необходимо знать типы значений, именуемых соответствующими идентификаторами;
3) идентификаторы, описанные в некоторой области действия, могут использоваться только в соответствии с описанием внутри этой области. Области действия идентификаторов введены в языки программирования для предоставления программисту возможности управления распределением памяти компьютера (области действия в языке Паскаль: программа, процедуры, функции).

Сформулируем контекстные условия, которые необходимо проверять при анализе программ.

1. В любой области действия без внутренних по отношению к ней областей действия никакой идентификатор не может быть описан более одного раза.

Для дальнейшего изложения нам потребуются следующие определения:

- **определяющим** является вхождение идентификатора в конструкцию, описывающую этот идентификатор;
- **прикладным** называется вхождение идентификатора в конструкцию, которая не является его описанием.

2. Каждому прикладному вхождению нестандартного идентификатора (стандартные идентификаторы — `integer`, `boolean`, `real`, `char`, `true`, `false` и др.) *должно найтись соответствующее ему определяющее вхождение.* Правило поиска определяющих вхождений называется **алгоритмом идентификации**, который заключается в следующем:

1) рассмотреть самую внутреннюю область действия, содержащую данное прикладное вхождение;
2) найти определяющее вхождение в рассматриваемой области действия. Если оно найдено, то процедура идентификации закончена, и данное прикладное вхождение идентификатора удовлетворяет контекстному условию. В противном случае — перейти к шагу (3);
3) найти область действия, непосредственно объемлющую только что рассмотренную. Если такая область найдена, то перейти на шаг (2). В противном случае процедура идентификации закончена, и так как определяющее вхождение не найдено, то данное прикладное вхождение идентификатора не удовлетворяет контекстному условию.

3. Контекстные условия предполагают также проверку соответствия типов величин, входящих в синтаксические конструкции программ; соответствия количества индексов у переменных с индексами и размерности соответствующих массивов и др.

6.2. Организация таблиц семантического анализатора

6.2.1. Таблица идентификаторов

Проверка контекстных условий требует знания атрибутов идентификаторов, используемых в программе. Анализатор получает эту информацию из описаний.

Способы использования идентификатора представим константами:

```
#define PROGS   300   /* ПРОГРАММА */
#define TYPES   301   /* ТИП */
#define CONSTS  302   /* КОНСТАНТА */
#define VARS    303   /* ПЕРЕМЕННАЯ */
#define PROCS   304   /* ПРОЦЕДУРА */
#define FUNCS   305   /* ФУНКЦИЯ */
```

Для хранения атрибутов идентификаторов воспользуемся структурой:

```
struct treenode
  {unsigned hashvalue; /* значение hash-функции */
   char *idname;       /* адрес имени в таблице имен */
   unsigned class;     /* способ использования
                          идентификатора */
   ?                   /* информация о типе */
  };
```

В дальнейшем **struct treenode** будет постепенно уточняться.

6.2.1.1. Таблица идентификаторов области действия (без внутренних по отношению к ней областей)

Для каждой области действия создается **таблица идентификаторов** (ТИ). Рассмотрим ее организацию. Когда начинается анализ некоторой области действия, соответствующая ТИ пуста. В процессе обработки объявлений для каждого нового идентификатора элемент добавляется в таблицу только один раз, но поиск в таблице ведется всегда, когда встречается идентификатор. Так как на этот процесс тратится много времени, важно выбрать такую организацию ТИ, которая допускала бы эффективный поиск и расширение. Подходящей структурой данных в этом случае является *дерево поиска с вершинами, связанными указателями*, где вершина дерева содержит описатель идентификатора.

Добавим поля:

```
struct treenode *leftlink, *rightlink;
```

в структуру **treenode**. Все вершины в ТИ упорядочим по значению функции **hash**.

Определим имя типа:

```
typedef struct treenode NODE;
```

Обработку определяющего вхождения идентификатора будем выполнять с помощью функции:

```
NODE *newident
     ( unsigned hashfunc  /* значение hash-функции
                             для идентификатора */,
       char *addrname     /* адрес имени
                             в таблице имен */,
       int classused      /* способ использования
                             идентификатора */)

/* Обработка определяющего вхождения идентификатора;
если идентификатор с заданным способом использования
ранее был включен в дерево, то результат — NULL,
иначе — ссылка на вершину дерева, соответствующую
этому идентификатору */
```

Поиск определяющего вхождения идентификатора для заданного прикладного вхождения будем выполнять с помощью функции:

```
NODE *searchident
     ( unsigned hashfunc /* значение hash-функции
                            для идентификатора */,
       char *addrname    /* адрес имени
                            в таблице имен*/,
       int *setofclass   /* множество возможных
                            способов использования
                            идентификатора */)

/*Обработка прикладного вхождения идентификатора —
поиск соответствующего ему определяющего вхождения.
Результат функции — указатель на вершину дерева,
соответствующую данному идентификатору*/
```

Пример 6.1. Таблица идентификаторов для программы

```
program example1;
  const  day = 10;
  type massiv = array [1..10]  of real;
  var  month, year : massiv;
begin          . . .          end.
```

представлена на рис. 6.1.

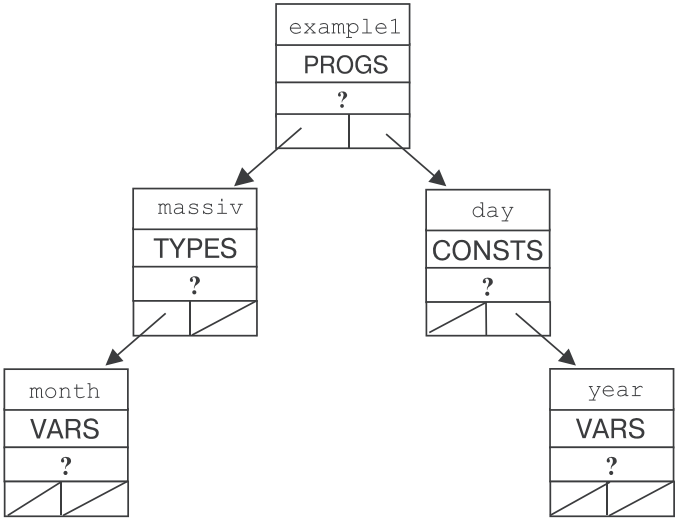


Рис. 6.1. Таблица идентификаторов для примера 6.1

Примечание: изображая дерево, соответствующее таблице идентификаторов, мы будем придерживаться следующих соглашений:

- hash-функция (признак упорядочения вершин) подобрана таким образом, что позволяет получить достаточно хорошо сбалансированное дерево; значение hash-функции на схеме не указывается;
- в вершине дерева вместо адреса имени в таблице имен мы будем изображать соответствующее имя.

6.2.1.2. Нейтрализация семантических ошибок

Если при обработке прикладного вхождения идентификатора не нашлось соответствующего определяющего вхождения,

то необходимо сформировать сообщение об ошибке и занести в ТИ неопределенный идентификатор с атрибутами, полученными из контекста. Эти действия предотвращают повторные сообщения о необъявленном идентификаторе. Поэтому значение функции **searchident** всегда определено.

Повторные сообщения могут возникать также из-за неправильного описания идентификатора. В этом случае в ТИ заносятся описатели всех прикладных вхождений этого идентификатора. Если использование идентификатора не соответствует описанию, то нужно просмотреть все его описания в ТИ, и если ранее встречалась такая некорректность, то формировать сообщение не следует; в противном случае в таблицу ошибок заносится информация о встретившейся ошибке, а в ТИ добавляется элемент с информацией о новом некорректном использовании идентификатора.

6.2.1.3. Информация о параметрах процедур (функций)

Определяющее вхождение имени процедуры (функции) обрабатывается функцией **newident**. Для проверки соответствия фактических и формальных параметров при вызове процедуры (функции) требуется **информация о типах и способах передачи параметров.** Поэтому к структуре **treenode** необходимо добавить следующее поле:

```
struct idparam *param    /* указатель на информацию
                            о параметрах */
```

Структура **idparam** описывается так:

```
struct idparam  /* информация о параметрах */
 {
  ? *typeparam  /* информация о типе параметра */;
  int mettransf /* способ передачи параметра */;
  struct idparam  *linkparam  /* указатель на информацию
                                 о следующем параметре */;
 } ;
```

Пример 6.2. Структура вершины-описателя процедуры с параметрами-значениями и параметром-переменной

```
procedure Y ( c, d : integer; var e : real );
  begin        . . .                    end;
```

изображена на рис. 6.2.

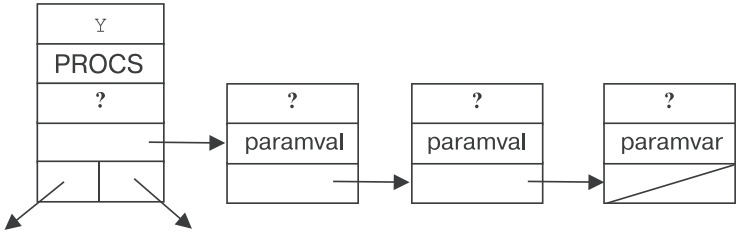


Рис. 6.2. Пример вершины-описателя процедуры

Для проверки соответствия формальных и фактических параметров параметра-процедуры (параметра-функции) к структуре **idparam** добавим поле:

```
struct prfun_as_par *par  /* указатель на
                     информацию о параметрах
                     параметра-процедуры (функции) */;
```

С целью повышения надежности программ в Паскале в качестве параметров нельзя использовать процедуры, фактические параметры которых передаются переменной, т. е. параметрами таких процедур могут быть только параметры-значения. Поэтому структура **prfun_as_par** принимает следующий вид:

```
struct prfun_as_par
  {
    ?  /* информация о типе параметра параметра -
          процедуры, ( параметра - функции ) */;
    struct prfun_as_par *linkpp  /* указатель на
                                    следующий элемент */;
  };
```

Пример 6.3. Структура вершины-описателя процедуры с параметром-процедурой и параметром-функцией

```
procedure W
 ( procedure a ( x, y :real );
   function b ( z : real; k : integer ) : char );
begin                . . .                    end;
```

представлена на рис. 6.3.

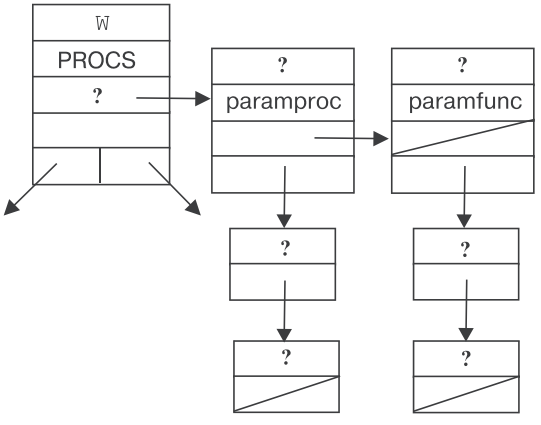


Рис. 6.3. Пример вершины-описателя процедуры

6.2.1.4. Таблица идентификаторов для программы (с вложенными областями действия)

Теперь рассмотрим организацию ТИ для программы в целом. Один и тот же идентификатор может быть описан и использован многократно в различных областях действия, и для каждого такого описателя должна найтись вершина дерева в ТИ соответствующей области действия. Согласно алгоритму идентификации, правило нахождения определяющего вхождения идентификатора состоит в том, чтобы сначала просмотреть ТИ текущей области действия, затем — непосредственно объемлющей и т. д., пока не будет найдено описание идентификатора или не завершится просмотр таблиц идентификаторов для всех объемлющих областей. Мы можем осуществить такой поиск, сохраняя адреса всех ТИ в *стеке*, элемент которого описывается структурой:

```
struct scope
  {
    struct treenode *firstlocal;    /* указатель на ТИ
                                       области действия */
    ?  *typechain;                /* указатель на таблицу
                                  типов области действия */
    ?  *labelpointer;             /* указатель на таблицу
                                  меток области действия */
    struct scope *enclosingscope;      /* указатель на
                        элемент стека области действия,
                    непосредственно объемлющей данную */
  };
typedef struct scope SCOPE;
```

Адрес вершины стека мы будем хранить во внешней переменной:

```
struct scope *localscope;
```

Пример 6.4. Таблица идентификаторов для программы

```
program example4;
{используем описания процедур без параметров}
    var desk, lamp: integer;
    procedure p1;
    var table: real; desk: integer;
        procedure p2;
        begin { p2 }   . . .   end;
    begin { p1 }       . . .       end;
begin          . . .                 end.
```

во время анализа процедуры **p2** показана на рис. 6.4.

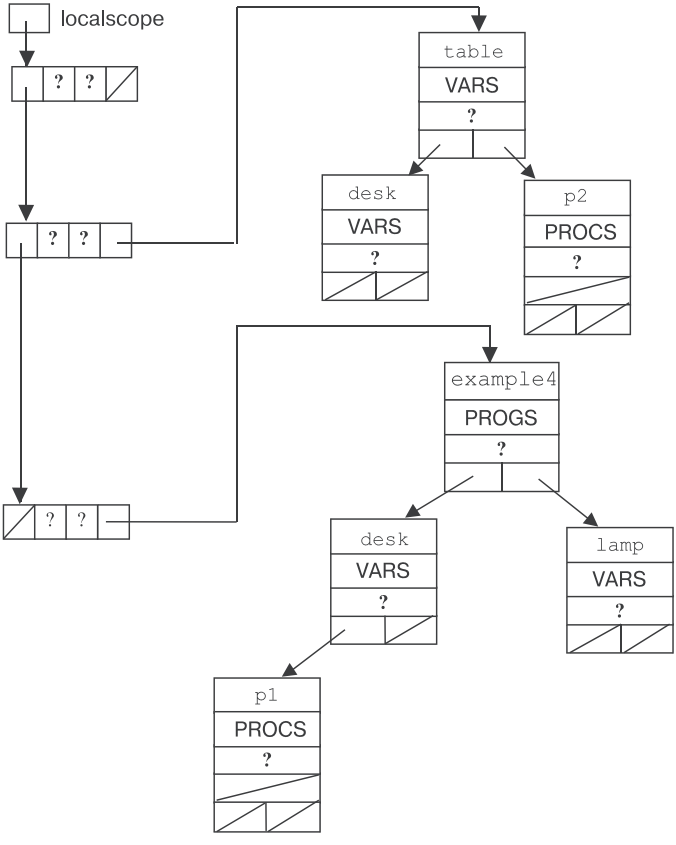


Рис. 6.4. Таблица идентификаторов для примера 6.4

Таким образом, начиная анализ новой области действия, необходимо создать для нее элемент стека. С этой целью мы воспользуемся функцией:

```
open_scope ( )
  /* создание элемента стека для
                    текущей области действия */
   {
     SCOPE *newscope  /* указатель на
                         новый элемент стека */;
     newscope = (SCOPE*) malloc (sizeof(SCOPE));
     newscope -> firstlocal = NULL;
     newscope -> typechain = NULL;
     newscope -> labelpointer = NULL;
     newscope -> enclosingscope = localscope;
     localscope = newscope;
   }
```

Анализ области действия следует завершить вызовом функции, которая удаляет таблицы этой области действия и соответствующую секцию стека:

```
close_scope ( )
  /* удаление таблиц текущей области действия */
  {
    SCOPE *oldscope   /* указатель на удаляемый
                                  элемент стека */;
    oldscope = localscope;
    localscope = localscope -> enclosingscope;
    /* удаление таблицы идентификаторов */
    dispose_ids (oldscope  -> firstlocal);
    /* удаление таблицы типов (см. 6.2.2) */
    dispose_types (oldscope -> typechain);
    /* удаление таблицы меток (см. 6.2.3) */
    del_list_lab (oldscope -> labelpointer);
    free (oldscope);
  }
```

6.2.1.5. Стандартные идентификаторы

В языке Паскаль некоторые идентификаторы заранее предопределены, т. е. распознаются компилятором без предварительного описания в программе. Их называют **стандартными**. Например, `false`, `true`, `maxint` — стандартные константы; `integer`, `boolean`, `real`, `char`, `text` — стандартные типы; `input`, `output` — стандартные файловые переменные. Кроме

того, существуют стандартные функции (abs, cos, eof, eoln, exp и др.) и стандартные процедуры (read, readln, write, writeln, new и др.). В отличие от ключевых слов, стандартные идентификаторы можно *переопределять*, так как предполагается, что они описаны в некоторой области действия, которая содержит основную программу. Назовем эту область действия *фиктивной*. В таблицу идентификаторов фиктивной области действия занесем информацию о стандартных идентификаторах типов, констант и переменных. В случае переопределения новое описание стандартного идентификатора всегда включается в таблицу идентификаторов текущей области действия. Следовательно, при обработке прикладного вхождения переопределенного идентификатора стандартное описание становится недоступным.

Пример 6.5. Таблица идентификаторов для программы

```
program example5;
{переопределяем стандартный идентификатор integer}
  var red, green: integer;
  procedure p1;
    var integer: real; blue: real;
    procedure p2;
    begin { p2 }
          integer:=red*green+ blue {правильно!}
    end;
  begin { p1 }     . . .       end;
begin            . . .                 end.
```

во время анализа тела процедуры **p2** показана на рис. 6.5.

Вернемся к проблеме обработки параметров. Так как параметры являются локальными переменными в области действия процедуры (функции), то информацию о них необходимо размещать в таблице идентификаторов этой процедуры (функции).

Пример 6.6. Таблица идентификаторов для программы

```
program example6;
{используем описание процедуры с параметрами}
 var yellow: integer;
 procedure work
     ( black: integer; var result: real );
  var colors : array [2..20] of integer;
  begin { work }     . . .       end;
begin            . . .                 end.
```

во время анализа тела процедуры **work** показана на рис. 6.6.

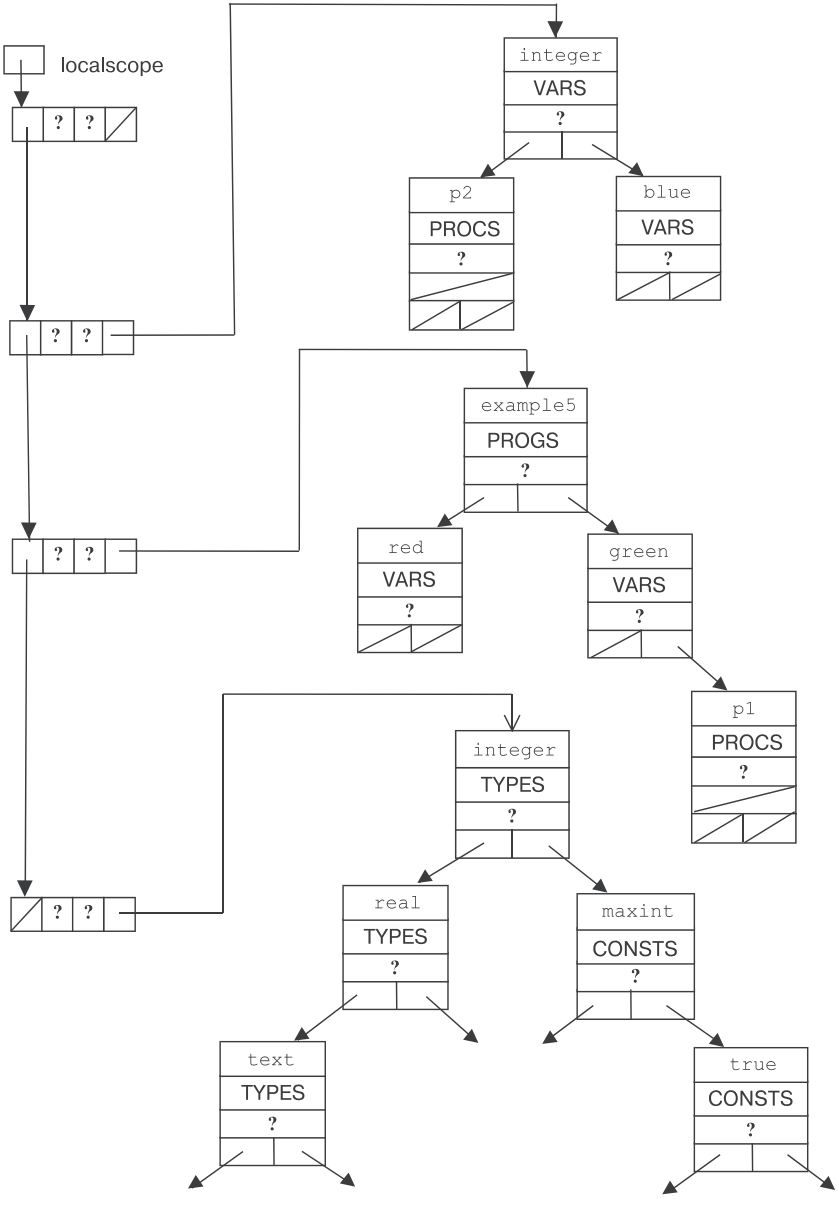


Рис. 6.5. Таблица идентификаторов для примера 6.5

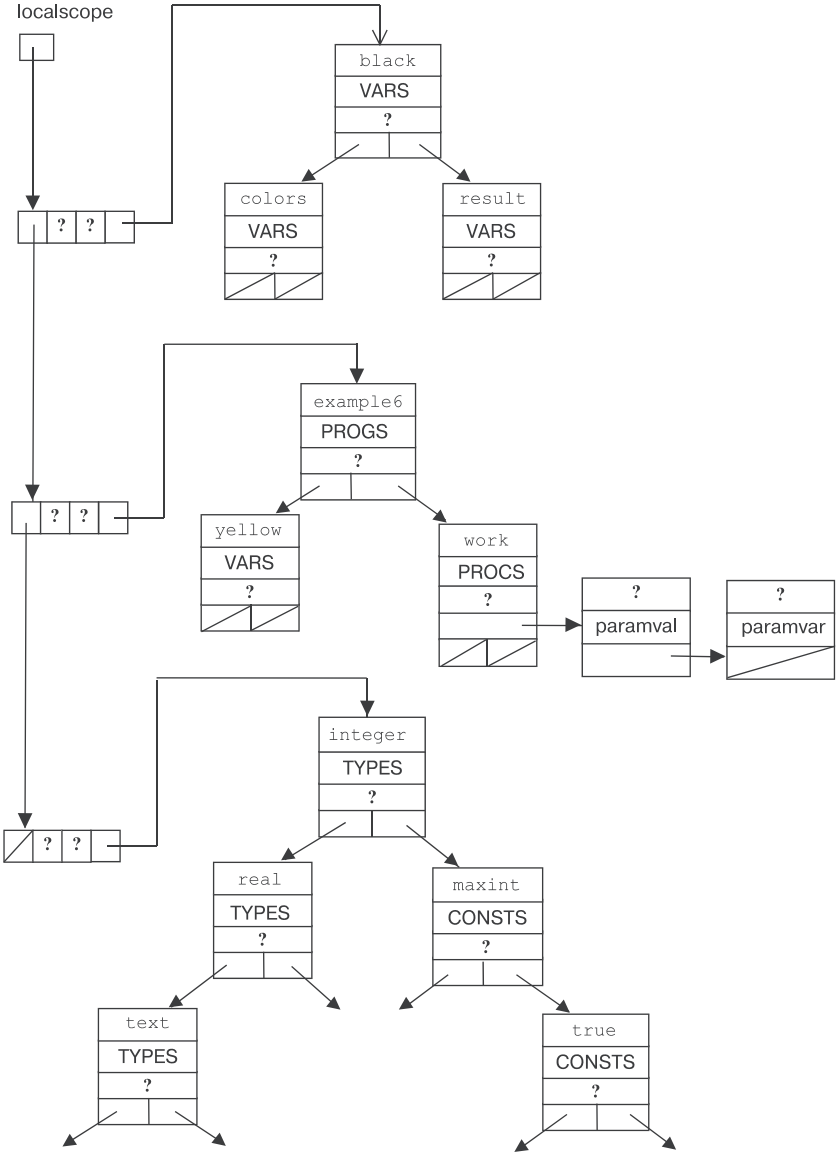


Рис. 6.6. Таблица идентификаторов для примера 6.6

6.2.1.6. Опережающие описания процедур

Вызов процедуры (функции) может предшествовать полному описанию процедуры (функции), если для этой процедуры (функции) дано *опережающее описание*.

Например:

```
procedure y (c,d: integer; var e : real);
        forward;
          ...
        y (...);  {вызов процедуры y }
          ...
  procedure y;
    begin   ...        end;
```

Список параметров здесь описывается только один раз в опережающем описании.

Для анализа описания процедуры необходимо знать, встречалось ли опережающее описание этой процедуры. Поэтому в структуру **treenode** добавим новое поле:

```
int forw  /* информация об опережающем описании;
             forw = TRUE, если встречалось
             опережающее описание;
             иначе forw = FALSE */
```

Если опережающее описание процедуры уже встречалось, то при обработке ее полного описания функция **newident** возвращает ссылку на вершину-описатель, созданную при опережающем описании.

Таким образом, описатель идентификатора в таблице идентификаторов принимает следующий вид:

```
struct treenode
 { unsigned hashvalue /* значение hash-функции */;
   char *idname        /* адрес имени в
                                 таблице имен */ ;
   unsigned class      /* способ использования */;
   ?                   /* информация о типе */;
   union
      {  /* для констант */
          union const_val constvalue  /* значение
                                         константы */;
         /* для процедур (функций) */
         struct
           { struct idparam  *param /* указатель
                     на информацию о параметрах */;
```

```
            int forw  /* информация об
                         опережающем описании */;
          } proc;
      } casenode;
  struct treenode *leftlink ;
  struct treenode *rightlink;
};
union const_val    /* значение константы */
 { int intval      /* целого или
                      символьного типа */;
    float realval   /* вещественного типа */;
    char *charval   /* перечислимого типа
                      (адрес в таблице имен) */
 };
```

Коротко о главном

1. Синтаксис языка программирования описывается с помощью формальных и неформальных правил.
2. Формальные правила описания синтаксиса — формы Бэкуса—Наура и синтаксические диаграммы.
3. Синтаксические правила языка программирования, которые задаются с помощью естественного языка (неформально), называются контекстными условиями.
4. Проверка контекстных условий требует знания атрибутов идентификаторов, используемых в программе. Анализатор получает эту информацию из описаний.
5. Атрибуты идентификаторов хранятся в специальной таблице идентификаторов (ТИ).
6. Подходящей структурой данных для таблицы идентификаторов одной области действия (без внутренних по отношению к ней областей) является дерево. Вершина дерева содержит описатель идентификатора.
7. Таблица идентификаторов для Паскаль-программы в целом организуется в виде стека. Каждый элемент стека содержит указатель на таблицу идентификаторов соответствующей области действия.
8. Для хранения информации о стандартных идентификаторах создается фиктивная область действия, непосредственно объемлющая основную программу.
9. Описание идентификатора всегда включается в таблицу идентификаторов текущей области действия (в вершину стека).

10. В языке Паскаль разрешается переопределять стандартные идентификаторы. При обработке прикладного вхождения переопределенного идентификатора стандартное описание недоступно.

Задание

1. Изобразите таблицу идентификаторов для программы:

```
program mytest;
const n=10;
type index = 1.. n;
     object = record v, w : integer end;
var  i: index;
     a: array [ index ] of object;
     w1, w2: integer;
procedure try (i: index, tw,av: integer;);
    const m=9;
    var av:integer;
    procedure try1 (v1,v2: real; var  z: real);
       type new = array [1..3,'a'..'f'] of index;
       var bz1: new;  a: char;
           w1, w2: text;
         begin       . . .      end {try1};
       begin        . . .        end { try }
    begin             . . .              end.
```

- во время анализа тела процедуры **try1**;
- во время анализа тела процедуры **try**;
- во время анализа операторов основной программы.

2. Ответьте на вопрос: как будет использоваться эта таблица для обработки прикладных вхождений идентификаторов?

6.2.2. Таблица типов

Информация о типе идентификатора должна храниться в таблице идентификаторов (см. **struct treenode**). Чтобы избежать дублирования этой информации для однотипных переменных, введем *дескриптор типа*, а в вершину ТИ поместим адрес этого дескриптора. Для единообразия обработки всех типов введем дескрипторы и для стандартных типов.

Таблица типов (ТТ) создается для каждой области действия. Когда начинается анализ области действия, соответствующая ТТ пуста. По мере обработки объявлений типов дескрипторы типов добавляются в ТТ. Для организации ТТ воспользуемся *односвязным списком*.

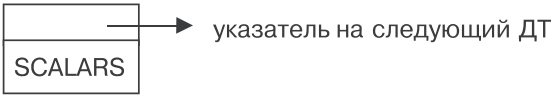
Введем константы для обозначения кодов типов:

```
#define SCALARS     401   /* стандартный
                             скалярный тип */
#define LIMITEDS    402   /* ограниченный тип */
#define ENUMS       403   /* перечислимый тип */
#define ARRAYS      404   /* регулярный тип
                             (массив) */
#define REFERENCES  405   /* ссылочный тип */
#define SETS        406   /* множественный тип */
#define FILES       407   /* файловый тип */
#define RECORDS     408   /* комбинированный тип
                             (запись)*/
```

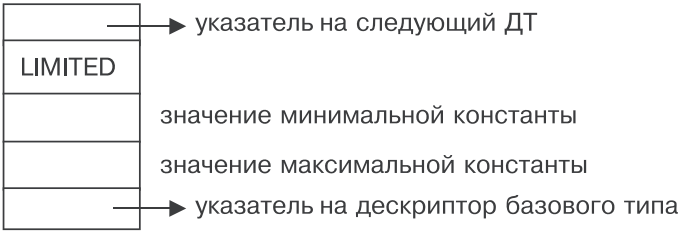
6.2.2.1. Структура дескрипторов типов

Сначала изобразим дескрипторы типов в виде схем, а затем опишем их на языке Си.

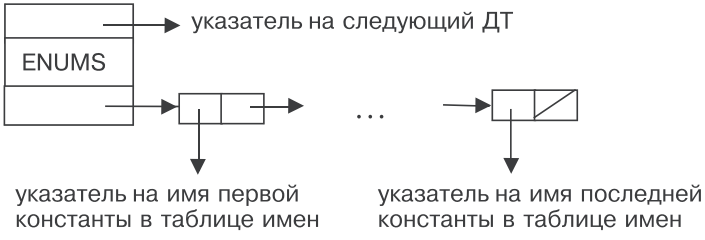
1. Стандартный скалярный тип



2. Ограниченный тип



3. Перечислимый тип

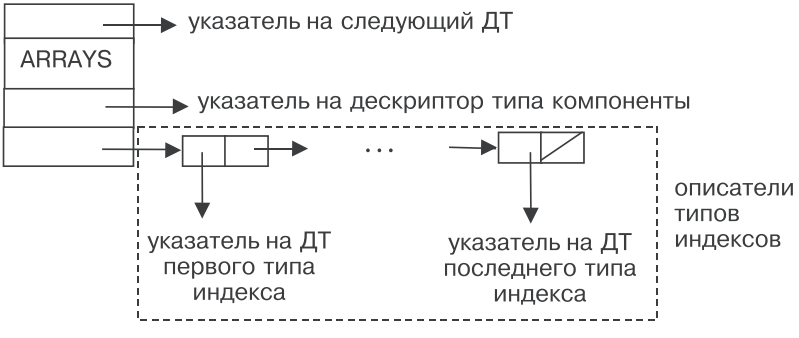


Пример 6.7. Таблица идентификаторов и таблица типов для программы

```
program example7;
{используем перечислимый и ограниченный типы}
  type number = ( one, two, three );
  var  b, a : number;  c: two.. three;
  begin                    ...        end;
```

представлена на рис. 6.7.

4. Регулярный тип (массив)



Пример 6.8. Таблица идентификаторов и таблица типов для программы

```
program example8;
{используем перечислимый, регулярный
                        и ограниченный типы}
type my = (one, two);
var a: my;  d: array [my, 1..10] of my;
begin                  ...              end.
```

изображена на рис. 6.8.

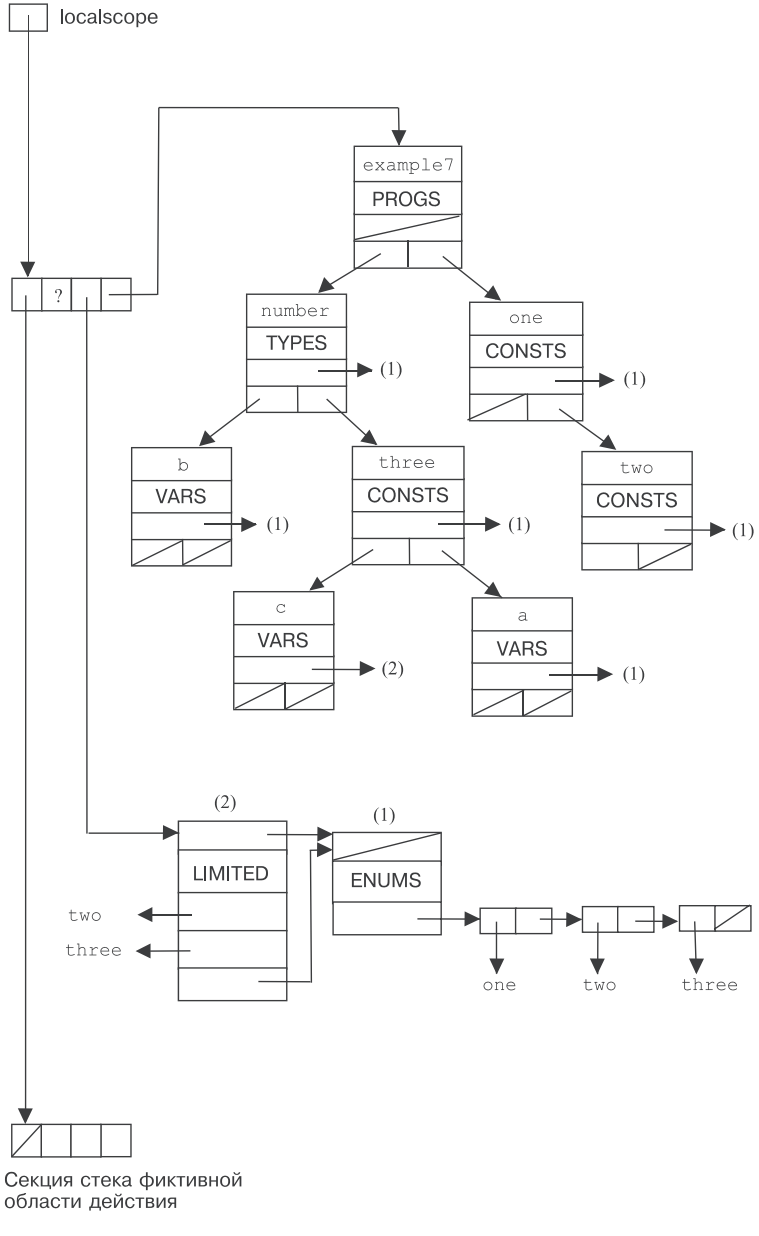


Рис. 6.7. Таблица идентификаторов и таблица типов для примера 6.7

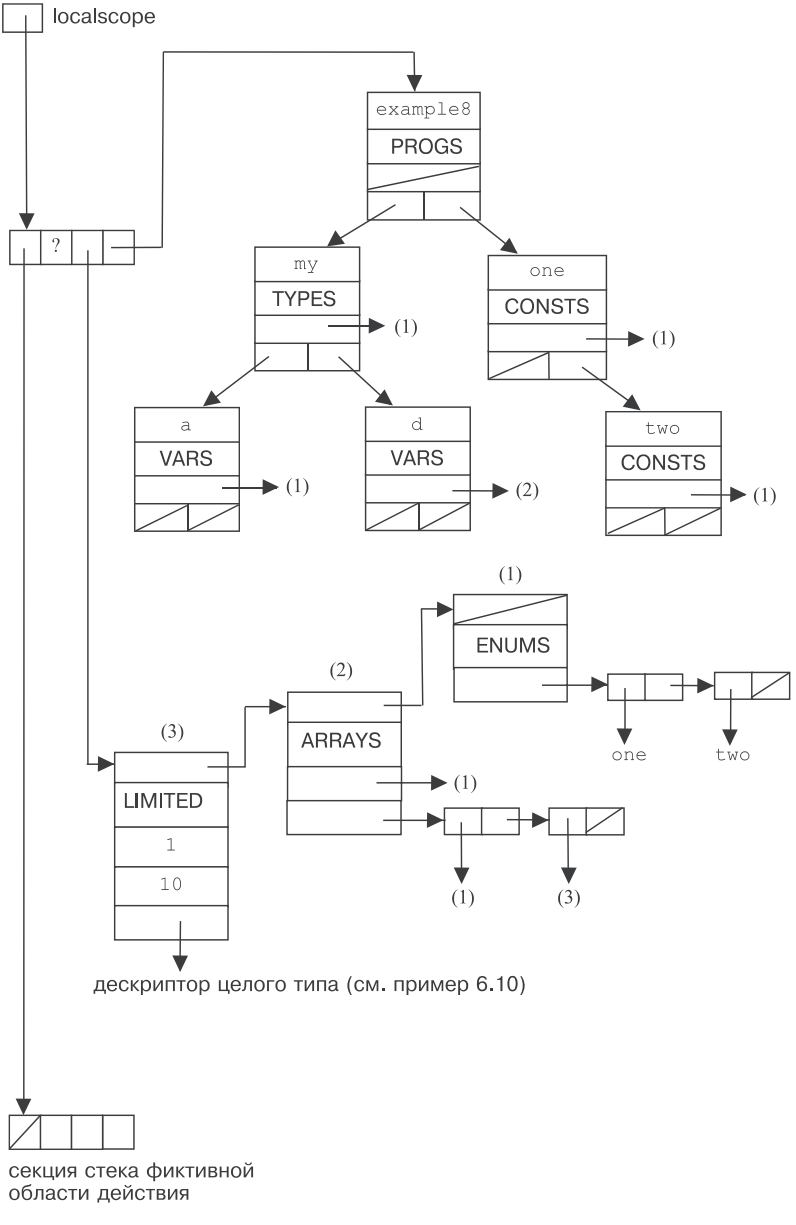
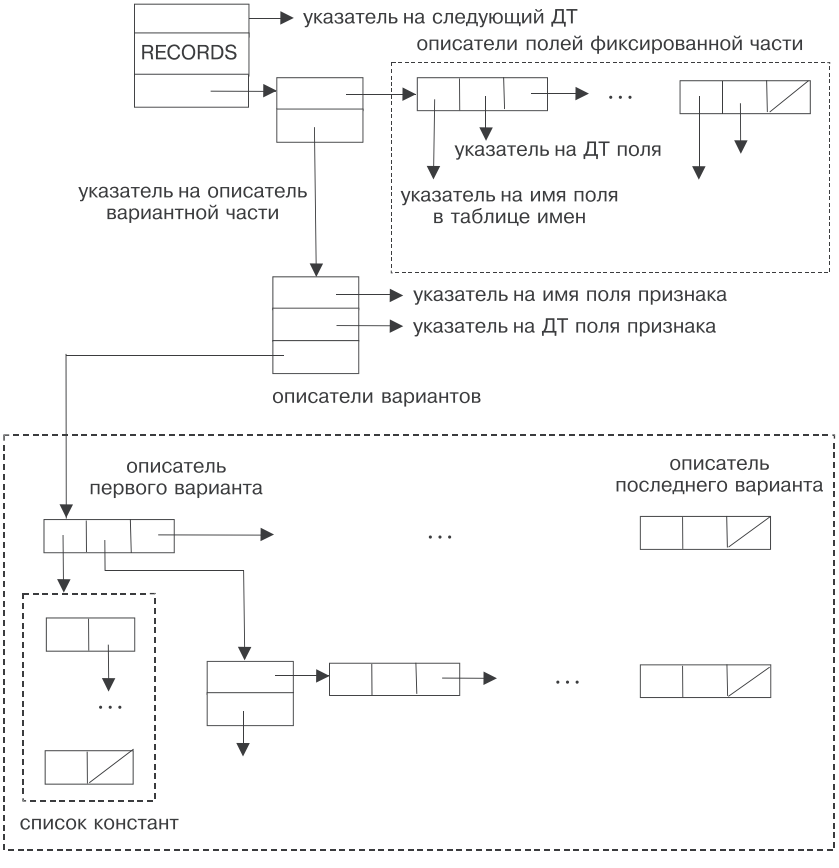


Рис. 6.8. Таблица идентификаторов и таблица типов для программы 6.8

5. Комбинированный тип (запись)



Пример 6.9. Таблица идентификаторов и таблица типов для программы

```
program example9;
{используем комбинированный и ограниченный типы}
type date = record month: 1..12; year: 1..2006
          end;
var now: date;
begin          . . .                          end.
```

представлена на рис. 6.9.

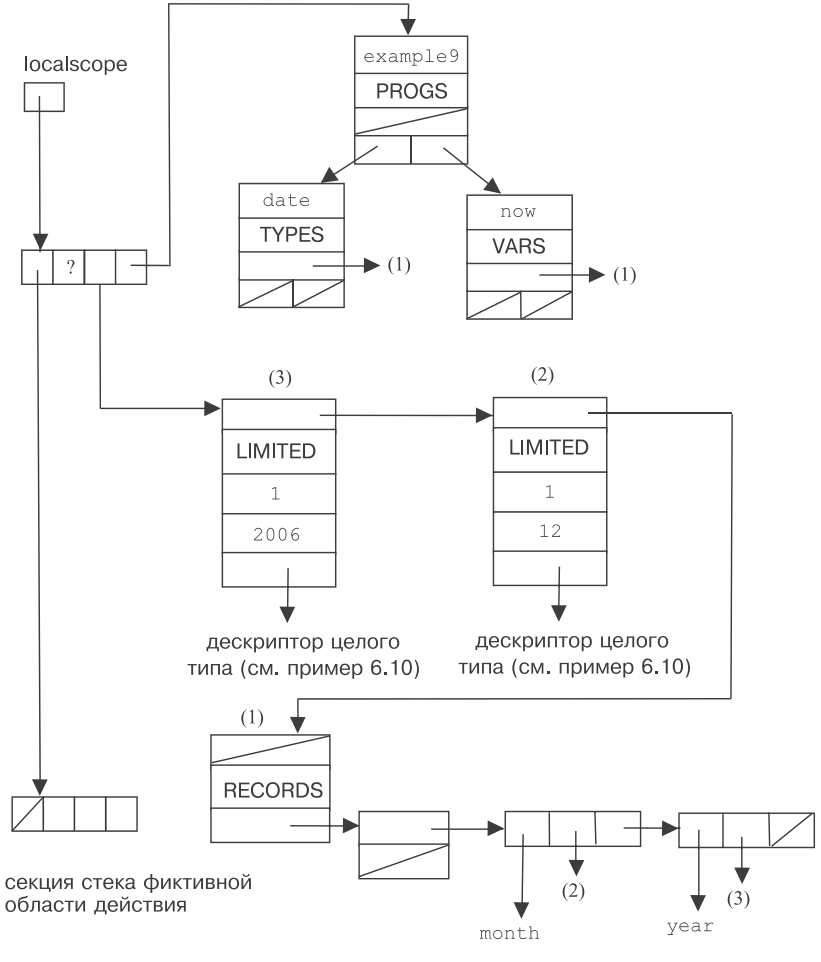
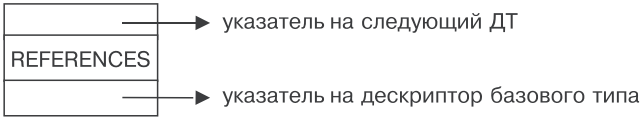
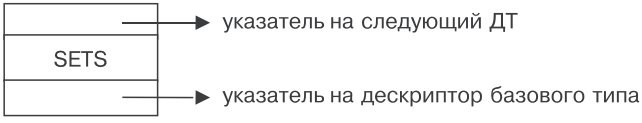


Рис. 6.9. Таблица идентификаторов и таблица типов для примера 6.9

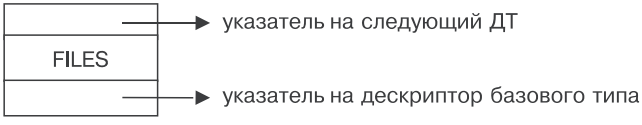
6. Ссылочный тип



7. Множественный тип



8. Файловый тип



Теперь опишем структуру дескриптора типа на С:

```
struct typerec
  {struct typerec *next;     /* указатель на следующий
                                    дескриптор типа */
   unsigned typecode;                /* код типа */
   union variapart casetype;     /* вариантная часть
                                    дескриптора */
  };

typedef struct typerec TYPEREC;

union variapart     /*вариантная часть дескриптора */
  { /* для множеств, файлов, указателей */
    TYPEREC *basetype;      /* указатель на дескриптор
                                    базового типа */
    struct lim            /* для ограниченного типа */
      { TYPEREC *basetype;            /* указатель на
                           дескриптор базового типа  */
        union                          /*минимальная и
                         максимальная константы ...:*/
          { struct { char*min,*max;} enptrs;
                               /*перечислимого типа */
            struct { int min, max;} intptrs;
                         /*целого и символьного типа */
          } diapason;
      } limtype;

    struct /* для массивов  */
      { struct indextyp *indextype;       /* указатель
                       на дескрипторы типов индексов */
```

```
      TYPEREC  *basetype;              /* указатель на
                          дескриптор типа компоненты */
    } arraytype;

  /* для перечислимого типа */
  struct reestrconsts  *firstconst;     /* указатель
                 на первый элемент списка констант */
  /* для записей */
  struct fieldreestr *fields;
};  /* end of union variant */

  struct reestrconsts            /* структура элемента
            списка констант перечислительного типа*/
   { char *addrconsts;         /* адрес идентификатора
                                      в таблице имен */
     struct reestrconsts *next;        /* указатель на
                                  следующую структуру*/
   };

  struct indextyp          /*описатель типа индекса */
   { TYPEREC  *Type;                    /* указатель на
                         дескриптор типа индекса в TT*/
     struct indextyp *next;             /* указатель на
                                 следующий описатель */
   };

  struct fix           /* описатель поля фиксированной
                                      части записи */
   { char *Name;              /* указатель на имя поля
                                      в таблице имен */
     TYPEREC  *Type;                    /* указатель на
                           дескриптор типа поля в  TT */
     struct fix *next;                  /* указатель на
                                 следующий описатель */
  };

  struct sig                          /*описатель признака
                                  вариантной части записи*/
   { char *Name;                /*указатель на имя поля
                                признака в таблице имен */
     TYPEREC  *Type;           /* указатель на дескриптор
                                типа поля признака в TT */
```

```
    struct caseField *variants;          /*указатель
                                на описатель вариантов*/
   };
  struct fieldreestr     /* описатель списка полей */
   { struct fix *fixedfields;          /* указатель на
              описатели полей фиксированной части */
     struct sig *sign;       /* указатель на описатель
                                    вариантной части */
   };

  struct caseField          /*  описатель варианта */
   { struct constchain *firstconst;      /* указатель
                                  на список констант */
     struct fieldreestr *fldlist;        /* указатель
                           на описатель списка полей*/
     struct caseField *next;             /* указатель на
                                  следующую структуру*/
 };
```

После введения дескриптора типа уточним запись об идентификаторе:

```
struct treenode
  {
   unsigned hashvalue    /* значение hash-функции */;
   char *idname      /* адрес имени в таблице имен */;
   unsigned class          /* способ использования */;
   TYPEREC *idtype; /* указатель на дескриптор типа */
   union
      { /* для констант */
        union const_val constvalue;          /* значение
                                            константы */
       /* для процедур (функций) */
        struct
         { struct idparam *param          /* указатель
                        на информацию о параметрах */;
           int forw                    /* информация об
                                опережающем описании */;
         } proc;
      } casenode;
   struct treenode *leftlink;
   struct treenode *rightlink;
  };
```

При описании типа идентификатора может быть допущена ошибка. Для подавления повторных сообщений об ошибках при обработке прикладных вхождений идентификаторов будем считать, что как неопределенный, так и ошибочный тип совместим с любым другим типом. Для неопределенного и ошибочного типов значение поля **idtype** в записи об идентификаторе равно **NULL**.

Дескрипторы стандартных типов будем хранить в таблице типов фиктивной области действия.

Пример 6.10. Организация таблицы типов для программы

```
program example10;
  {учитываем
    1. существование таблицы типов фиктивной
       области действия
    2. наличие ошибки в описании типа}
      var one, two : integer;
        three : real;
        four : (alfa, digit);
        five : array [2 .. 10] of seven {ошибка!};
 begin                ...              end.
```

изображена на рис. 6.10.

Внешние переменные **chartype**, **inttype**, **realtype**, **booltype**, **texttype** содержат адреса дескрипторов для типов **char**, **integer**, **real**, **boolean** и **text** соответственно.

Таблица типов программы в целом имеет организацию, аналогичную ТИ, так как типы локализованы в той области действия, в которой описаны.

В дальнейшем нам потребуется функция **newtype**, которая создает и присоединяет новый дескриптор типа к таблице типов:

```
TYPEREC *newtype (int tcode  /* код типа */ )
 {
  struct typerec *new; /* указатель  на
                                  дескриптор типа */
  new =
 (struct typerec*)malloc(sizeof(struct typerec));
  new -> typecode = tcode;
  new -> next = localscope -> typechain;
  switch ( new -> typecode )
```

```
    { case LIMITEDS:
          new -> casetype.basetype = NULL;
      case SCALARS:
          break;
      case ENUMS:
          new -> casetype.firstconst = NULL;
          break;
      case SETS:
      case FILES:
      case REFERENS:
          new -> casetype.basetype = NULL;
          break;
      case ARRAYS:
       new -> casetype.arraytype.basetype = NULL;
       new -> casetype.arraytype.indextype = NULL;
       break;
      case RECORDS:
          new -> casetype.fields = NULL;
    }
  localscope -> typechain = new;
  return (new);
  }
```

6.2.3. Таблица меток

Для каждой области действия создается **таблица меток** (ТМ). Перед началом анализа области действия соответствующая таблица меток пуста. Для каждого определяющего вхождения метки элемент добавляется в таблицу только один раз, но поиск в ней ведется всякий раз, когда встречается описание метки. При повторном описании метки формируется сообщение об ошибке.

Для организации ТМ используется односвязный список. Элемент этого списка описывается структурой:

```
struct labellist
   {
     int meaning                    /* значение метки */;
     struct labellist *nextlabel          /* указатель на
                          следующий элемент списка */;
   };
```

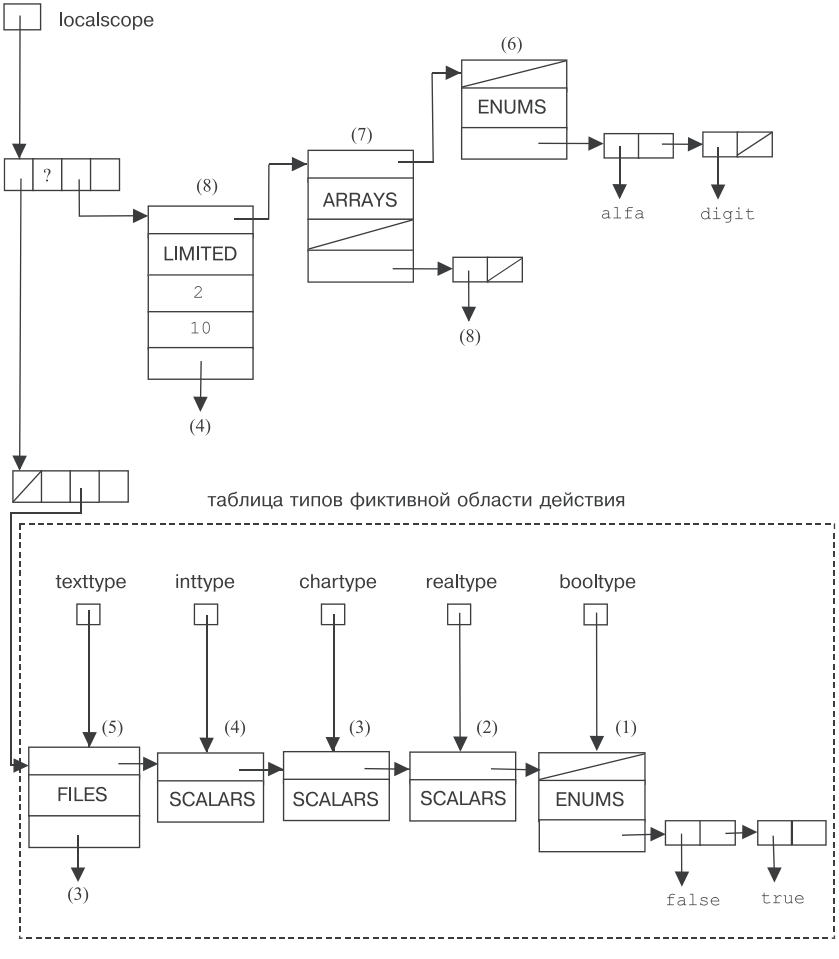


Рис. 6.10. Таблица типов для примера 6.10

Пример 6.11. Таблица меток для программы

```
program example11;
   label 1,20,40;
   var one, two : integer;
      three : real;
      four : (alfa, digit);
   begin          ...    end.
```

представлена на рис. 6.11.

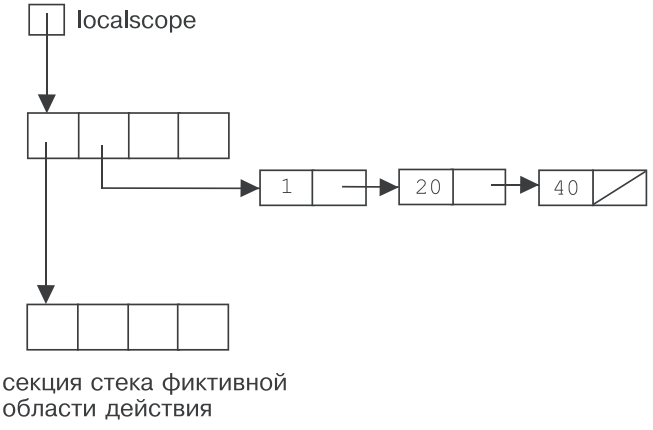


Рис. 6.11. Таблица меток для примера 6.11

Принимая во внимание таблицу меток, уточним описание структуры **scope**:

```
struct scope                    /* элемент стека вложенных
                                        областей действия*/
{
  struct treenode *firstlocal; /* указатель на ТИ */
  TYPEREC  *typechain;          /* указатель на ТТ */
  struct labellist *labelpointer; /* указатель на ТМ */
  SCOPE *enclosingscope;      /* указатель на элемент
                              стека области действия,
                    непосредственно объемлющей данную */
};
```

Коротко о главном

1. Информация о типе хранится в дескрипторе типа.
2. Для каждой области действия создается таблица типов. По мере обработки объявлений типов их дескрипторы добавляются в эту таблицу.
3. Для организации таблицы типов удобно использовать односвязный список.
4. Информация о типе идентификатора в таблице идентификаторов — указатель на дескриптор типа.

5. Для неопределенного или ошибочного типа значение указателя на дескриптор типа равно **NULL**. Это соглашение используется для подавления повторных сообщений об ошибках при обработке прикладных вхождений идентификаторов.
6. Дескрипторы стандартных типов хранятся в таблице типов фиктивной области действия.
7. Для каждой области действия создается таблица меток. Для организации таблицы меток также используется односвязный список.

Задания

1. Изобразите таблицу идентификаторов и таблицу типов для программы:

```
program asd;
  const  n=10;
  type mas = array [(a,b,c), 'a'..'z']
                          of array [2..30] of real;
  var desk: record
              first: mas; second: ^integer;
              third: mas
            end;
      i, j: boolean;  f: integer;
  procedure fast (x: integer; var y:real);
    var k: char;
    begin  ...   y:= x+k*k {ошибка !};   ...   end;
  begin   ...
    mas := 6; {ошибка!}
    desk. first := desk.third;
    while i < j  do
      begin ...
        desk. second^ := 1.2 + f {ошибка!} ...
      end;
    fast ( f*f, f*r1, k); { ошибка! }
          ...
end.
```

2. Ответьте на вопрос: каков алгоритм обработки прикладных вхождений идентификаторов?

6.3. Программирование семантического анализатора

Дополним функции синтаксического анализа действиями, выполняющими семантический анализ. По мере того как синтаксический анализатор распознает некоторую конструкцию языка, он выполняет и семантическую обработку этой конструкции. Такая параллельная работа синтаксического и семантического анализаторов имеет место в однопроходных компиляторах.

6.3.1. Создание фиктивной области действия

```
void programme ( )
/* создание элемента стека для фиктивной
                              области действия */
{
 open_scope ( );
 /* построение ТИ и ТТ фиктивной области
                                      действия */
 /* см. рис. 6.12 */
 booltype = newtype ( ENUMS );
 search_in_table ("false");

 /* построение вершины в ТИ для константы false */
 entry = newident (hashresult, addrname, CONSTS);
 entry -> idtype = booltype;

 /* создание элемента списка констант в дескрипторе
     перечислимого типа  */
 (booltype -> casetype).firstconst =
                              newconst ( addrname );
 /* далее - аналогичные действия
                             для константы true */
 search_in_table ("true");
 entry =  newident (hashresult, addrname, CONSTS);
 entry -> idtype = booltype;
 (booltype -> casetype).firstconst -> next =
                            newconst ( addrname );

 /* продолжаем строить дескрипторы
                          стандартных типов*/
 realtype = newtype (SCALARS);
 chartype = newtype (SCALARS);
```

```
inttype = newtype (SCALARS);
texttype = newtype (FILES);
(texttype -> casetype).basetype = chartype;

/* заносим  в таблицу имен и ТИ остальные
                    стандартные идентификаторы  */
search_in_table ( "integer" );
entry = newident (hashresult, addrname, TYPES);
entry -> idtype = inttype;
search_in_table ("maxint");
entry = newident (hashresult, addrname, CONSTS);
(entry -> casenode).constvalue.intval = MAXCONST;
/* значение константы  MAXCONST зависит
     от конкретной вычислительной машины */
entry -> idtype = inttype;
search_in_table ("boolean");
...
search_in_table ("real");
entry = newident (hashresult, addrname, TYPES);
entry -> idtype = realtype;

/* создание элемента стека для области
                действия основной программы */
open_scope;
   /* синтаксический анализ */
accept (programsy);  accept (ident);
accept (leftpar);  accept (ident);
while (symbol == comma)
  {  nextsym ( );
     accept (ident);
  }
accept (rightpar);  accept (semicolon);
block ( );  accept (point);
}
```

Примечание: стандартные идентификаторы процедур и функций не хранятся в ТИ фиктивной области действия, а обрабатываются особым образом.

Результат инициализации таблицы идентификаторов и таблицы типов показан на рис. 6.12.

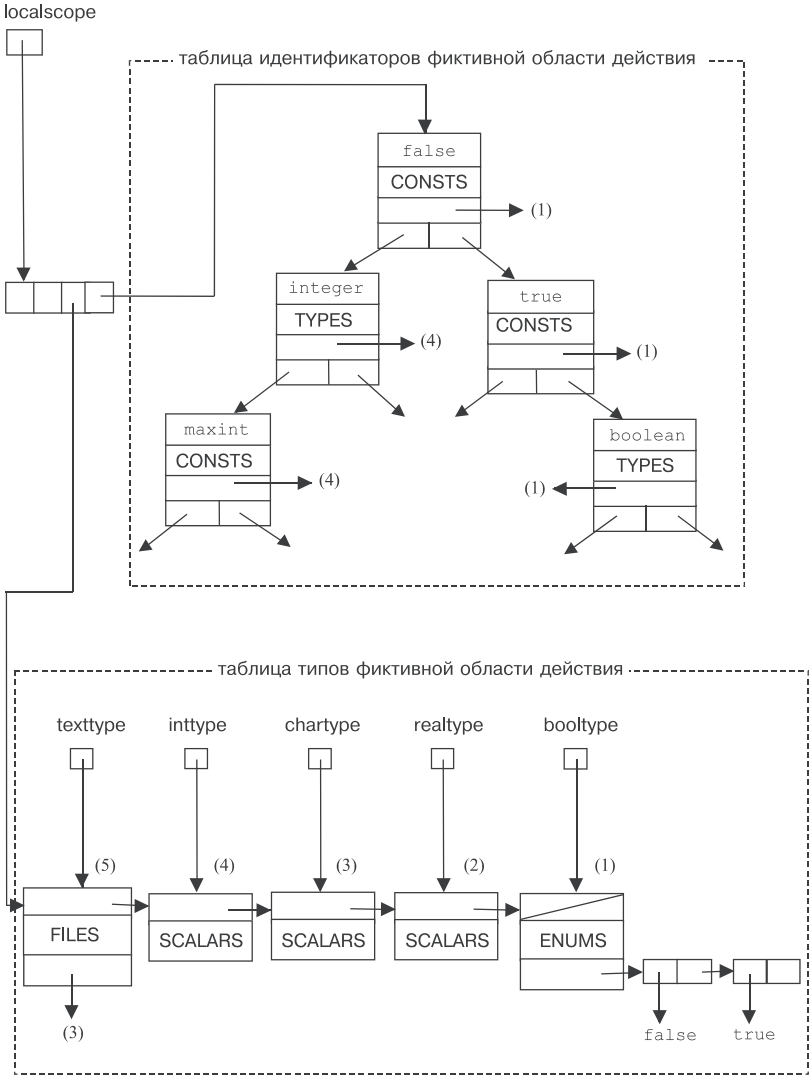


Рис. 6.12. Результат инициализации таблицы идентификаторов и таблицы типов

6.3.2. Анализ описания переменных

В процессе анализа конструкции *<описание однотипных переменных>* для каждого идентификатора в ТИ должна быть создана вершина типа **NODE**. Однако в момент формирования этой вершины поле **idtype** остается неопределенным, так как информация о типе идентификатора еще не прочитана анализатором. Поэтому необходимо построить вспомогательный список, содержащий адреса вершин ТИ для однотипных переменных (значение внешней переменной **varlist** — адрес начала этого списка).

Пример 6.12. Состояние ТИ и вспомогательного списка к моменту, когда анализатор обработал входной поток:

```
var
    k ,a ,b : integer;
```

до символа `integer` изображено на рис. 6.13.

После того как будет прочитана информация о типе, легко определить значение поля **idtype** для вершин, адреса которых содержатся во вспомогательном списке.

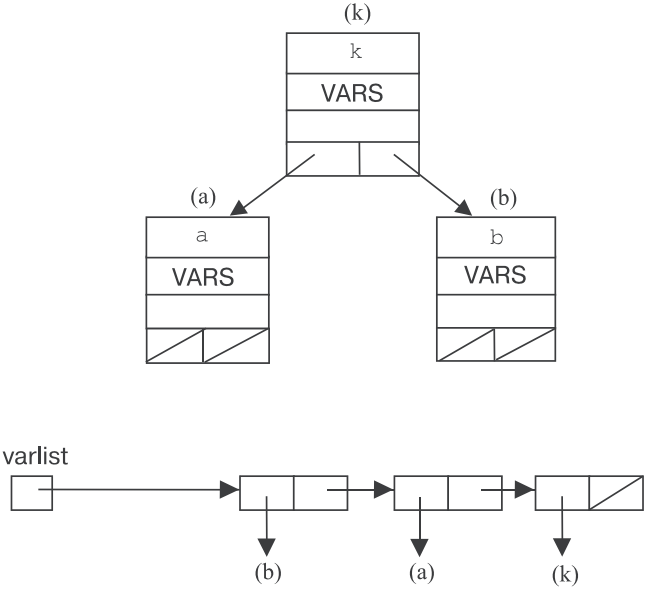


Рис. 6.13. Состояние ТИ и вспомогательного списка (пример 6.12)

Опишем тип элемента вспомогательного списка:

```
struct listrec
   { struct treenode *id_r;          /* адрес вершины
                                таблицы идентификаторов */
     struct listrec *next;         /* адрес следующего
                                    элемента списка */
   };
```

Теперь можно уточнить описание функции **vardeclaration**. Для этого воспользуемся двумя вспомогательными функциями — **newvariable** и **addattributes**.

```
void newvariable ( )
/* создание элемента вспомогательного списка ;
   в момент вызова этой функции информация о типе
   идентификатора еще не прочитана анализатором */
  {struct listrec *listentry;             /* указатель
                                   на текущий элемент
                              вспомогательного списка */
  if (symbol == ident)
   {listentry = (struct listrec*)
                       malloc(sizeof(struct listrec));
    listentry -> id_r =
                    newident(hashresult,addrname,VARS));
    listentry -> next = varlist;
    varlist = listentry;
   }
}

void addattributes ( )
/* присваивает значение полю idtype для вершин,
   адреса которых содержатся во вспомогательном
                                          списке */
  { struct listrec
    *listentry,  /* указатель на вершину
                       вспомогательного списка */
    *oldentry;   /* указатель на вершину,
                       предшествующую текущей
                    во вспомогательном списке */
      listentry = varlist;
      while (listentry != NULL)
```

```
      { listentry -> id_r -> idtype = vartype;
              /* внешняя переменная vartype содержит
               адрес дескриптора типа для однотипных
                                          переменных */
        oldentry = listentry;
        listentry = listentry -> next;
        free (( void*) oldentry);
      }
  }
```

Примечание: в дальнейшем для краткости изложения начальные и заключительные действия по нейтрализации синтаксических ошибок будут опущены.

```
void vardeclaration (unsigned *followers)
/* анализ конструкции
  <описание однотипных переменных> */
  {
    varlist = NULL;
    newvariable( );
    accept (ident);
    while (symbol == comma)
      {  nextsym ( );
         newvariable ( );
         accept (ident);
      }
    accept(colon);
    /* анализ конструкции  <тип>; результат
       функции type — адрес дескриптора типа */
    vartype = type (followers);        /* внешняя
                переменная vartype содержит адрес
                              дескриптора типа для
                          однотипных переменных */
    addattributes ( );
  }
```

В процессе семантического анализа описаний других конструкций языка, например опережающих описаний ссылок, также используются вспомогательные списки.

Пример листинга программы с сообщениями об ошибках, обнаруженных синтаксическим и семантическим анализаторами в разделе объявления переменных:

Работает Pascal-компилятор

```
  1  program example;
  2   var a,b: integer;
  3       a,b,, c:real;
**01**     ^ ошибка код  101
****** имя описано повторно
**02**      ^ ошибка код  101
****** имя описано повторно
**03**        ^ ошибка код   2
****** должно идти имя
  4      f: abc;
**04**       ^ ошибка код  104
****** имя не описано
  5      e: (one, two, three));
**05**                          ^ ошибка код  6
****** запрещенный символ
  6      myrecord :
  7           record
  8            1, first: 75..45;
**06**         ^ ошибка код  19
****** ошибка в списке полей
**07**                           ^ ошибка код  112
****** недопустимый ограниченный тип
  9             second : myrecord
**08**                     ^ ошибка код  100
****** использование имени не соответствует описанию
 10           end;
 11
 12   begin     end.

Компиляция окончена: ошибок - 8!
```

6.3.3. Анализ описания типов

В процессе анализа описаний типов, введенных пользователем, необходимо строить **дескрипторы типов.** Эти дескрипторы используются для анализа прикладных вхождений идентификаторов, проверки типов индексов у индексированных переменных и др.

В качестве примера рассмотрим семантический анализ *перечислимого типа*. Описание языка Паскаль содержит правило:

<простой тип> ::= *<перечислимый тип>* | *<ограниченный тип>* | *<имя типа>*

Поэтому анализ описания перечислимого типа мы будем выполнять в функции, соответствующей конструкции <*простой тип*>:

```
TYPEREC  *simpletype (unsigned *followers)
  /* анализ конструкции  <простой тип> */
{
TYPEREC *typentry;                        /* указатель на
                                       дескриптор типа*/
NODE *identry;          /* указатель на вершину в ТИ*/
struct reestrconsts *constlist=NULL;     /* указатель
                          на элемент списка констант
                    в дескрипторе перечислимого типа*/
switch (symbol)
 { case leftpar:                        /* анализ описания
                                   перечислимого типа */
     typentry=newtype(ENUMS);                /* строим
                         "каркас" дескриптора типа */
     do {
        nextsym( );
        if(symbol==ident)
          {
           /* создаем вершину в ТИ для
              константы перечислимого типа */
           identry=newident(hashresult,
                            addrname,CONSTS);
           /* запоминаем указатель на дескриптор
              типа константы в ТИ */
           identry->idtype=typentry;
           /* запоминаем имя константы в ТИ */
           (identry->casenode).constvalue.enumval=
                                        addrname;
            if(constlist==NULL)
            /* создаем первый элемент списка
                константа в дескрипторе
                перечислимого типа */
            (typentry->casetype).firstconst=
                    constlist=newconst(addrname);
              else
              /* создаем следующий элемент списка
                 константа в дескрипторе
                 перечислимого типа */
              { constlist->next=
                            newconst(addrname);
```

```
                    constlist=constlist->next;
                  }
              }
            accept(ident);
          }
      while(symbol==comma)
      accept(rightpar);
      break;
    case ident: . . .
    default: . . .
 }
return(typentry);
}
```

Теперь рассмотрим, как воспользоваться построенным дескриптором типа и информацией в таблице идентификаторов.

Пусть в Паскаль-программе встречаются описания:

```
type day = ( Monday, Tuesday, Wednesday, Thursday,
                  Friday, Saturday, Sunday );
     workday1 = Monday .. Friday;
     myday = Sunday ..  Saturday; {ошибка!}
var  x: integer;
```

и оператор `x := Friday {ошибка!}`. При определении ограниченного типа должно выполняться условие: минимальная константа предшествует максимальной константе в соответствии со списком перечисления. Выполнение этого условия может быть проверено с помощью списка констант в дескрипторе типа: сначала в списке ищется минимальная константа, а затем просмотр списка продолжается для нахождения максимальной константы. В результате таких действий и будет найдена ошибка в типе **myday**.

Оператор присваивания `x:= Friday` записан неверно, так как переменной целого типа нельзя присваивать значение перечислимого типа. Чтобы обнаружить эту ошибку, необходимо воспользоваться таблицей идентификаторов. Для переменной и константы семантический анализатор найдет вершины в этой таблице, определит адреса их дескрипторов типов и выполнит проверку на соответствие типов.

Рассмотренный пример демонстрирует, как можно создать и использовать дескриптор перечислимого типа. Работа с дескрипторами остальных типов, конечно же, имеет свои особенности, но не должна вызывать больших трудностей, так как во многом выполняется аналогично.

6.3.4. Анализ операторов

Сначала опишем функцию анализа для конструкции *<оператор>*:

```
void statement (unsigned *followers)
  /* анализ конструкции  <оператор> */
 { if (symbol == intc)
    { /* семантический анализ метки */
       ...
       nextsym ( );
       accept ( colon );
     }
    switch (symbol)
     { case ident:
         if ( идентификатор — имя поля,
              переменной или функции)
           /* анализ оператора присваивания */
            assignment ( followers );
           else
           /* анализ вызова процедуры */
              call_proc (followers);
        break;
       case beginsy:
            compoundstatement (followers); break;
       case ifsy:
            ifstatement (followers); break;
       case whilesy:
            whilestatement (followers); break;
       case repeatsy:
            repeatstatement (followers); break;
       case forsy:
            forstatement (followers); break;
       case casesy:
            casestatement (followers); break;
       case withsy:
            withstatement (followers); break;
       case gotosy:
            gotostatement (followers); break;
       case semicolon:
       case endsy:
       case untilsy:
       case elsesy: break; /* в случае
                           пустого оператора */
     }
 }
```

Рассмотрим анализ условного оператора:

```
void ifstatement (unsigned  *followers)
 {
   TYPEREC *exptype;                    /* указатель на
                          дескриптор типа выражения */
   unsigned ptra [SET_SIZE];              /* множество
                                    внешних символов */
   nextsym ( );
   /* формирование множества внешних символов
      для конструкции <выражение> */
   set_disjunct (af_iftrue, followers, ptra);
   exptype = expression (ptra);
   /* проверка типа выражения */
   if (!compatible (exptype, booltype))
        error (135);  /* тип операнда должен быть
                                              boolean */
   accept ( thensy );
   /* формирование  множества внешних символов
      для конструкции <оператор> */
   set_disjunct (af_iffalse, followers, ptra);
   statement (ptra);
   if (symbol == elsesy)
      {
        nextsym ( );
        statement (followers);
      }
 }
```

В результате семантического анализа операторов цикла с пост-условием и пред-условием формируется сообщение об ошибке, если тип выражения не является логическим.

Анализ оператора цикла с параметром включает проверку следующих контекстных условий:

- параметр цикла должен быть любого скалярного (кроме вещественного) или ограниченного типа;
- типы управляющей переменной и граничных выражений должны быть совместными.

```
void forstatement (unsigned *followers)
/* анализ конструкции  <цикл с параметром> */
{
 TYPEREC *exptype;                      /* указатель на
                           дескриптор типа выражения */
 TYPEREC *vartype;                      /* указатель на
                     дескриптор типа параметра цикла */
```

```
unsigned ptra [SET_SIZE],          /*множество внешних
                                                 символов */
          codevar;              /* код типа переменной */
nextsym();
/* формирование множества внешних символов
   для конструкции <параметр цикла> */
set_disjunct (af_forassign, followers, ptra);
vartype = name_parameter(ptra);
if (vartype != NULL)
 {
  codevar = vartype -> typecode;
  if (codevar != SCALARS && codevar != ENUMS
      && codevar != LIMITEDS
      || vartype == realtype )
    error(143);                          /* недопустимый
                                  тип параметра цикла */
 }
accept(assign);
/* формирование множества внешних символов
для первого выражения*/
set_disjunct (af_for1, followers, ptra);
exptype = expression (ptra );
if (!compatible(vartype, exptype))
      error(145);     /* конфликт типов */
if ((symbol == tosy) || (symbol == downtosy))
   nextsym();
  else error (55);                /* должно идти
                          слово to или downto */
   /* формирование множества внешних символов
      для второго выражения*/
set_disjunct (af_whilefor, followers, ptra);
exptype = expression( ptra );
 if (!compatible (vartype, exptype))
      error(145);       /* конфликт типов */
 accept (dosy);
 statement (followers);
}
```

Некоторые проблемы возникают в процессе анализа оператора присоединения. При определении типа «запись» имена полей могут совпадать с именами переменных. Так как внутри оператора присоединения имена переменных-записей опускаются, то

возникает вопрос: что обозначает имя внутри оператора **with** — поле или переменную? В языке Паскаль эта проблема решается так: предпочтение отдается именам полей записи. Если две переменные из списка переменных-записей имеют поля, обозначенные одним и тем же идентификатором, то внутри оператора присоединения этот идентификатор обозначает поле той переменной, которая указана в списке позже.

В процессе анализа оператора **with** строится стек указателей на дескрипторы типов переменных-записей, содержащихся в заголовке этого оператора. Элемент стека описывается структурой:

```
struct withstack
  {
    struct typerec *varptr;            /* указатель на
                                       дескриптор типа
                                  переменной - записи */
    struct withstack *nextwith;          /*  указатель
                             на следующий элемент стека */
  }

  typedef struct withstack WITHSTACK;
```

Этот стек является общим для вложенных операторов присоединения программы (процедуры/функции). Перед началом обработки раздела операторов очередной области действия стек пуст. Новый элемент добавляется на вершину стека для каждой переменной-записи из заголовка оператора **with**. По завершении анализа очередного оператора присоединения из стека удаляется такое количество элементов, сколько переменных-записей перечислено в заголовке этого оператора.

Пример 6.13. Построение стека для оператора присоединения.

Пусть имеются описания переменных:

```
var r1: record
          a ,b ,c : integer
        end;
      r2: record
            a: real; b: boolean;
            r3: record
                  c: char; d: integer
                end
          end;
```

и оператор присоединения:

```
with r1 do
   with r2, r3 do
      < оператор >
```

Стек указателей на дескрипторы типов переменных-записей после анализа заголовка внутреннего оператора **with** изображен на рис. 6.14.

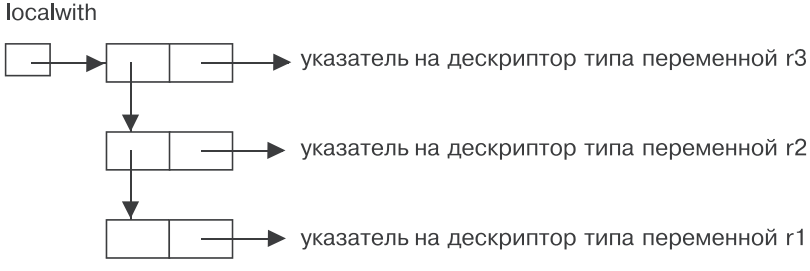


Рис. 6.14. Стек указателей на дескрипторы типов переменных-записей

При анализе прикладного вхождения идентификатора в первую очередь проверяется условие нахождения его внутри оператора присоединения (**localwith != NULL**). Если это условие выполнено, то можно предположить, что идентификатор является именем поля. Чтобы определить тип этого поля, необходимо найти запись, которой оно принадлежит. Для этого просматривается стек указателей на дескрипторы типов записей. В случае удачного поиска из дескриптора типа записи определяются атрибуты идентификатора. Если ни один из дескрипторов не содержит такого поля, то делается предположение, что идентификатор — это имя переменной, и поиск продолжается в ТИ.

Пример 6.14. Использование стека переменных-записей.

Пусть задано описание переменных (из примера 6.13) и оператор присоединения:

```
with r1 do
   begin
      with r2, r3 do
         begin  a:= 10.5;   b:= true;
                c:= '8';    d:= 5
         end;
     a:= 10;  b:= 15; c:= 20;
     b:= true {ошибка!}
  end
```

Воспользуемся стеком, изображенным на рис. 6.14. В процессе анализа внутреннего оператора **with** поля **a** и **b** будут найдены в дескрипторе типа переменной **r2**, а поля **c** и **d** — в дескрипторе типа переменной **r3**. По завершении внутреннего оператора **with** с вершины стека удаляются два элемента. Теперь идентификаторы **a**, **b** и **c** трактуются как поля переменной **r1** и, следовательно, для оператора **b:=true** необходимо выдать сообщение об ошибке:

```
void withstatement (unsigned *followers)
  /* анализ конструкции  < оператор присоединения> */
  {
    TYPEREC *vartype;                    /* указатель на
                          дескриптор типа переменной */
    WITHSTACK *withptr;                  /* указатель на
                                текущий элемент стека */
    unsigned
      var_count,          /* счетчик числа переменных -
                  записей в операторе присоединения */
      ptra [SET_SIZE],                      /* множество
                                     внешних символов */
      i,                  /* вспомогательный счетчик */
                        /* формирование внешних символов
                         для конструкции <переменная> */
    set_disjunct (af_with, followers, ptra);
 /* af_with - множество символов, ожидаемых сразу
    после анализа переменной в заголовке
    оператора with — "," и "do" */
    var_count = 0;
    do
     {
       nextsym();
       vartype = variable (ptra);
       if (vartype != NULL)
         if (vartype -> typecode != RECORDS)
           error (140); /* переменная не есть запись */
         else
           {
             /* создание нового элемента стека */
             withptr = (WITHSTACK*)
                        malloc(sizeof(WITHSTACK));
             withptr -> varptr = vartype;
             withptr -> nextwith = localwith;
```

```
            localwith = withptr;
            varcount++;
          }
      }
    while (symbol == comma);
    accept (dosy);
    statement (followers);
    /* из стека удаляется такое количество
       элементов, сколько переменных перечислено
       в заголовке оператора присоединения */
    for (i = 0; i < var_count; i++)
      {
       withptr = localwith;
       localwith = localwith -> nextwith;
       free (withptr);
      }
  }
```

Пример листинга программы с сообщениями об ошибках, обнаруженных синтаксическим и семантическим анализаторами в разделе описаний и операторе присоединения:

Работает Pascal-компилятор

```
   1  program example;
   2  var
   3        x,:integer;
**01**       ^ ошибка код   2
****** должно идти имя
   4        st:100..10;
**02**                ^ ошибка код  112
****** недопустимый ограниченный тип
   5    day:(Monday, Tuesday, Wednesday, Thursday,
   6         Friday, Saturday, Sunday);
   7    work: Monday .. Friday;
   8    my: Sunday ..  Saturday;
**03**                         ^ ошибка код  112
****** недопустимый ограниченный тип
   9
  10    k:record
  11       one:integer;
  12       two,three:;
**04**               ^ ошибка код   10
****** ошибка в типе
```

```
 13       case kind: char of
 14            'l':(name:char;
 15                 date:integer;
 16                 one:integer);
**05**              ^ ошибка код  101
****** имя описано повторно
 17            'y':(country: char)
 18     end;
 19   b3: record
 20        number:char;
 21        marka:record
 22                u: integer;
 23                s:char;
 24               end
 25      end;
 26
 27 begin    x:= Friday;
**06**                  ^ ошибка код  145
****** конфликт типов
 28        day := Wednesday;
 29        with k,b3 do
 30           begin
 31              s:=b3.marka.s;
**07**              ^ ошибка код  104
****** имя не описано
 32              marka.s := 'q';
 33              k:='s';
**08**                    ^ ошибка код  145
****** конфликт типов
 34              k :=st;
 35              one := 5;
 36              k.date:=12;  date:= 100;
 37              a.a:=10;  number := 'f'
**09**              ^ ошибка код  104
****** имя не описано
 38        end;
 39     number:='f';   k.date:=12; my := Sunday;
**10**   ^ ошибка код  104
****** имя не описано
 40 end.
Компиляция окончена: ошибок -  10 !
```

6.3.5. Анализ выражения

В процессе семантического анализа выражения проверяется:

- способ использования идентификатора (например, в выражении нельзя использовать имя процедуры или имя типа);
- совместимость типов операндов для каждой операции;
- соответствие формальных и фактических параметров в вызове функции;
- количество и типы индексов у индексированных переменных и др.

Рассмотрим функции синтаксического и семантического анализа для конструкций *<выражение>*, *<простое выражение>*, *<слагаемое>* и *<множитель>*. В этих функциях выполняется проверка типов операндов и способов использования идентификаторов.

```
TYPEREC *   expression(unsigned *followers)
  /* анализ конструкции <выражение> */
{
 TYPEREC *ex1type=NULL,
         *ex2type; /* указатели на дескрипторы
                      типов простых выражений */
 unsigned
   after_simexpr[SET_SIZE],       /* внешние символы */
   operation;                  /* операция над простыми
                                  выражениями*/
 /* формирование внешних символов
                              для простого выражения */
 set_disjunct (followers, op_rel, after_simexpr);
 /* анализ простого выражения*/
 ex1type = simple_expression (after_simexpr);
 if (belong(symbol, op_rel))           /* найден знак
                                 операции отношения */
   {operation = symbol;      /* запоминаем операцию */
    nextsym ( );
    ex2type = simple_expression(followers);
    /* проверка совместимости типов
                                  простых выражений */
    ex1type = comparing(ex1type, ex2type, operation);
   }
  return(ex1type);
}
```

```
TYPEREC* simple_expression (unsigned *followers)
/* анализ конструкции <простое выражение> */
{
 TYPEREC *ex1type,
         *ex2type;        /* указатели на дескрипторы
                                  типов  слагаемых */
 unsigned
  after_term[SET_SIZE],          /* внешние символы */
  sign=0,      /* имеется ли знак у 1-го слагаемого */
  operation;    /* операция над слагаемыми: +, -, OR */
  if (symbol == minus || symbol == plus)
        {sign = 1; nextsym ( );}
  /* формирование внешних символов для слагаемого */
  set_disjunct (op_add, followers, after_term);
  /* анализ слагаемого */
  ex1type = term (after_term);
  /* проверка - правильно ли записан знак
    перед первым слагаемым? */
  if (sign )  right_sign (ex1type);
  while(belong(symbol, op_add))
    {
      operation = symbol;  nextsym();
      ex2type = term(after_term);
      /* проверка совместимости типов слагаемых */
     ex1type=test_add(ex1type,ex2type,operation);
    }
  return( ex1type );
}


TYPEREC*   term (unsigned *followers)
 /* анализ конструкции <слагаемое> */
{
  TYPEREC *ex1type,
          *ex2type; /* указатели на дескрипторы
                              типов множителей */
  unsigned
     after_factor[SET_SIZE],    /* внешние символы */
     operation;          /* операция над множителями:
                                 *, /, DIV, MOD, AND */
  /* формирование внешних символов для множителя */
   set_disjunct (followers, op_mult, after_factor);
```

```
    /* op_mult - внешняя переменная - указатель на
            битовую строку для множества символов:
                              *, /, and, div, mod */
 /* анализ множителя */
 ex1type = factor(after_factor);
 while (belong(symbol, op_mult))
   {
     operation = symbol;  nextsym();
     ex2type = factor(after_factor);
     /* проверка совместимости типов множителей */
     ex1type = test_mult(ex1type,ex2type,operation);
   }
  return(ex1type);
}


TYPEREC * factor(unsigned  *followers)
 /* анализ конструкции <множитель> */
{
  TYPEREC *exptype;                    /* указатель на
                          дескриптор типа множителя */
  NODE *node;               /* указатель на вершину ТИ,
          соответствующую текущему идентификатору */
  unsigned
   after_express[SET_SIZE];  /* внешние символы */
  switch (symbol)
 {
  case leftpar:                            /* выражение
                                  в круглых скобках */
     nextsym();
     /* формирование внешних символов
                                         для выражения*/
     set_disjunct(followers, rpar, after_express );
     exptype=expression( after_express );
     accept(rightpar);
     break;
   case lbracket:          /* конструктор множества */
     exptype=set(followers);
     break;
   case notsy:               /* отрицание множителя */
     nextsym ( );
     exptype = factor(followers);
```

```
  exptype = logical(exptype);      /* проверка типа
                                        множителя */
  break;
case intc:                      /*  целая константа */
  exptype = inttype;  nextsym();
  break;
case floatc:           /* вещественная константа */
  exptype = realtype;   nextsym();
  break;
case charc:             /*  символьная константа */
  exptype = chartype;   nextsym();
  break;
case nilsy:   /* константа nil */
  exptype = niltype;  nextsym();
  break;
case ident:
  if /* находимся в контексте оператора with и
                        идентификатор - имя поля */
   (localwith !=NULL
            &&  search_field() !=NULL)
  /* функция search_field ищет имя поля,
       используя стек указателей на дескрипторы
                    типов переменных-записей */
       exptype = variable(followers);
     else
       { /* ищем имя в ТИ */
       node = searchident(hashresult,
                      addrname, set_VARCONFUNCS );
       if (node != NULL)         /* нашли имя в ТИ */
       /* по способу использования имени
          выбираем путь дальнейшего анализа */
       switch (node-> class)
         {
           case VARS:          /*  имя переменной */
             exptype=variable(followers);
             break;
           case CONSTS:          /* имя константы */
             exptype = node -> idtype;
             nextsym();
             break;
           case FUNCS:           /* вызов функции*/
             exptype = CallFunc(followers,node);
```

```
                break;
              default: /* ошибка */
                exptype = NULL;
          }
              else /* имя отсутствует в ТИ */
               exptype = StandardFunc (followers);
                         /* имя стандартной функции
                                          или ошибка */
       }
     break;
   }
  return (exptype);
}
```

А теперь опишем функцию **right_sign**, которая проверяет, правильно ли записан знак перед слагаемым (**right_sign** вызывается в функции **simple_expression**):

```
unsigned  right_sign
  (TYPEREC *type            /* указатель на дескриптор
                                   типа слагаемого */)
         /* знак ( + или - )  может быть записан перед
               операндом, если его тип является целым
       или вещественным, или ограниченным на целом */
{
  if (type == NULL || type == inttype
      || type == realtype
      || ( type->typecode == LIMITEDS
      && (type->casetype).limtype.basetype == inttype))
    return (1);
  else
    {
      error(211);      /*недопустимые типы операндов
                                 операции + или - */
      return(0);
    }
}
```

Функции проверки совместимости типов **comparing**, **test_add**, **test_mult** и **logical** описываются аналогично.

Коротко о главном

1. В однопроходных компиляторах синтаксический и семантический анализаторы работают параллельно.
2. Создание таблицы идентификаторов и таблицы типов фиктивной области действия выполняется в функции, соответствующей конструкции *<программа>*.
3. В процессе анализа конструкции *<описание однотипных переменных>* необходимо строить вспомогательный список. Этот список содержит адреса вершин таблицы идентификаторов, для которых информация о типе еще не прочитана анализатором.
4. Семантический анализ описания типа, введенного пользователем, предполагает создание дескриптора типа.
5. Семантический анализ операторов цикла **while** и **repeat**, а также условного оператора состоит в проверке типа выражения (а именно: выражение должно быть логического типа).
6. Семантический анализ цикла с параметром включает в себя проверку типа управляющей переменной, а также соответствие типов управляющей переменной и выражений в заголовке оператора.
7. В процессе семантического анализа оператора присоединения необходимо учитывать, что внутри этого оператора предпочтение отдается именам полей записи, а не переменным.
8. Семантический анализ выражения включает проверку соответствия типов операндов для каждой операции выражения, способа использования идентификатора, количества и типов индексов у индексированных переменных и др.

Задания

1. Сформулируйте контекстные условия, которые необходимо проверять при анализе:
   - описаний простых типов, массивов, записей, множеств, файлов;
   - описаний процедур и функций;
   - операторов выбора, присваивания и вызова процедуры;
   - прикладного вхождения переменной.

2. Дополните синтаксический анализатор действиями по семантическому анализу.
3. Разработайте набор тестов для тестирования семантического анализатора.
4. Выполните тестирование анализатора, содержащего действия по нейтрализации синтаксических и семантических ошибок.

Глава 7

Введение в генерацию кода

Генерация кода — это машинно-зависимая часть компилятора, так как она определяется архитектурой конкретной вычислительной машины. Понятие *архитектуры* включает в себя все то, что предоставляет компьютер пользователю, программирующему на уровне машинных команд. Другими словами, архитектура определяет структуру оперативной памяти, организацию памяти процессора (*регистры*), способы *адресации* (определения местоположения операндов), способы представления данных и команды.

Таким образом, согласно классической схеме компиляции, для реализации языка программирования на N вычислительных машинах с различной архитектурой необходимо создать один анализатор и N генераторов кода.

Результат генерации кода — **объектная программа** — может представлять собой последовательность машинных команд или программу на языке ассемблера. Подход, при котором выполняется перевод исходной программы в программу на языке ассемблера (которая, в свою очередь, обрабатывается ассемблером), увеличивает общее время, требуемое для получения программы на машинном языке. В этом случае обработка результата работы компилятора ассемблером может занимать время, сравнимое со временем компиляции. Поэтому в качестве объектного кода мы выберем машинный язык.

Результатом работы промышленных компиляторов является **объектный модуль**, который содержит:

- константы;
- машинные команды;
- символические имена других программ, к которым он обращается;
- имена своих точек входа, к которым можно обращаться из других программ;
- возможно — другую, дополнительную информацию.

В дальнейшем объектный модуль объединяется с другими модулями и загружается в память компьютера для выполнения.

Разработчик генератора кода должен решить такие проблемы, как:

- выбор промежуточного представления программы;
- организация оперативной памяти во время выполнения программы;
- распределение регистров;
- выбор команд.

Архитектура широко используемых компьютеров (VAX-11, IBM PC) имеет большой семантический разрыв с принципами организации языков высокого уровня [7]. Этот семантический разрыв порождает ряд существенных проблем, связанных с разработкой программного обеспечения, а именно высокую стоимость, ненадежность и большой объем программ, а также сложность операционных систем и компиляторов. Разработчик генератора кода, имея в своем распоряжении достаточно примитивный набор машинных команд, вынужден генерировать, а машина — исполнять большое число команд. Один из возможных путей преодоления семантического разрыва — создание архитектуры, ориентированной на языки высокого уровня. В этом случае организация данных и набор машинных команд приближены к организации данных и набору операций нескольких языков высокого уровня. Для машин с подобной архитектурой процесс генерации кода существенно упрощается. *Модульный конвейерный процессор* (МКП), созданный в Институте точной механики и вычислительной техники им. С. А. Лебедева, относится к вычислительным системам, в архитектуру которых заложены средства поддержки конструкций ЯВУ (блочно-процедурный механизм, циклы, механизм исключительных ситуаций и др.). Поэтому, чтобы показать, как реализуются задачи генерации кода, мы воспользуемся архитектурой МКП.

Глава 8

Архитектура модульного конвейерного процессора

Рассмотрим **архитектуру модульного конвейерного процессора (МКП)** с точки зрения разработчика компилятора, а именно познакомимся с организацией памяти процессора (регистрами), структурой оперативной памяти, способами адресации, способами представления данных и командами. (При описании архитектуры МКП использованы материалы Института точной механики и вычислительной техники им. С. А. Лебедева [6].)

8.1. Регистры

Регистровую память МКП составляют пять групп регистров: скалярные, сумматоры, адресные, дисплеи и управляющие. В дальнейшем используется следующая система обозначений группы регистров: *Р* — скалярные, *С* — сумматоры, *Д* — дисплеи, *А* — адресные, *У* — управляющие. Конкретный регистр при этом обозначается идентификатором, состоящим из буквы и индекса, где начальная буква идентификатора определяет группу регистра, а индекс — это порядковый номер в группе. Нумерация регистров производится восьмеричными числами, начиная с нуля. Например, *С0* — сумматор с номером 0, а *Д37* — дисплей-регистр с номером 37. Нумерация двоичных разрядов любого регистра производится справа налево последовательными десятичными числами, начиная с нуля.

Все эти регистры, за исключением некоторых управляющих, явно адресуемы в командах.

Скалярные регистры. В эту группу входит 128 Р-регистров, длина каждого из которых равна 64 двоичным разрядам. Номера 0–3 соответствуют сумматорам, а остальные — *Р*-регистрам, которые используются для обработки чисел с плавающей точкой, целых чисел, логических значений и других операций.

Дисплей-регистры. Имеющиеся 32 64-разрядных *Д*-регистра хранят только дескрипторы сегментов памяти и определяют доступное адресное пространство программы.

Адресные регистры. Каждый из 16 адресных регистров имеет длину 37 разрядов. *А*-регистр состоит из двух полей — индекса и номера дисплея. Поле индекса занимает младшие 32 разряда, а поле номера дисплея — старшие 5 разрядов.

Индексные поля *А*-регистров независимы и образуют самостоятельную группу регистров, называемых далее «*индексными регистрами*», или *И*-регистрами. Конкретный *И*-регистр идентифицируется номером соответствующего *А*-регистра. Индексные регистры используются для обработки целочисленных данных — индексов.

Управляющие регистры. В эту группу входит большое количество регистров различной длины. Ввиду их разнообразного назначения, в МКП отсутствует единый механизм работы с *У*-регистрами.

8.2. Способы представления данных

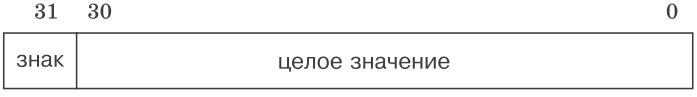
Числа с плавающей точкой представляются в *словном формате*. **Слово** состоит из трех полей — характеристики, знака и коэффициента:

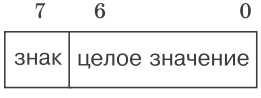
Здесь показатель степени *П* равен значению характеристики (*X*) – 1024. Разряд знака (*З*) содержит «0» для неотрицательных чисел или «1» — для отрицательных. Коэффициент (*К*) как для положительного, так и для отрицательного числа представляется в прямом коде. Мантисса вычисляется по формуле $M = K * 2^{-52}$, где *К* — двоичное значение коэффициента. В результате число с плавающей точкой вычисляется следующим образом: $(-1)^{З} * (K * 2^{-52}) * 2^{X-1024}$, где *З*, *К* и *X* — двоичные значения соответствующих полей.

Если какая-нибудь операция невыполнима для заданных значений операндов (например, если делитель равен нулю), то результатом такой операции в аппаратуре МКП будет значение «неопределенность», сопровождаемое соответствующим сигналом прерывания. Специальное число с плавающей точкой — *неопределенность* — имеет $K = 0$, $X = 0$ и $З = 1$.

Для чисел с плавающей точкой в МКП реализованы бинарные и унарные операции: сложение, вычитание, умножение, деление, изменение знака, получение абсолютного значения, выделение целой части с представлением результата в формате числа с плавающей точкой, выделение целой части с представлением результата в словном формате целого числа.

Целые числа. В соответствии с диапазоном представляемых значений, различаются стандартные и короткие целые числа. Стандартное целое число занимает 32 двоичных разряда, а короткое — 7 двоичных разрядов. Стандартные целые числа представляются в словном и *полусловном* формате, а короткие — только в коротком формате:





Словный формат используется для представления чисел, выбираемых из *С*- и *Р*-регистров. Полусловный формат используется для представления чисел, выбираемых из *И*-регистров. В коротком формате представляются целые числа — непосредственные данные в командах. Знаковый разряд равен «0» для положительных целых чисел и «1» — для отрицательных, при этом положительные целые числа представляются в прямом коде, а отрицательные — в дополнительном. При представлении стандартного целого числа в словном формате старшие разряды слова должны иметь нулевые значения.

Для целых чисел в МКП реализованы следующие операции: преобразование в число с плавающей точкой, сложение, вычитание, умножение, а также две группы специальных арифметических операций.

Двоичные числа являются машинным представлением неотрицательных целых чисел. Двоичные числа представляются в двух форматах: словном и коротком. В словном формате двоичное число занимает все слово, и, следовательно, двоичные числа находятся в диапазоне $0 \le Д \le 2^{64} - 1$. При этом двоичные целые числа в диапазоне от 0 до $(2^{31} - 1)$ совпадают по представлению с положительными целыми числами. Короткий формат двоичных чисел используется для задания непосредственных данных. Короткие двоичные числа находятся в диапазоне $0 \le КД \le 127$.

Операции над двоичными числами: сложение, циклическое сложение, вычитание.

Битовые наборы — это произвольные последовательности двоичных значений, которые представляются в словном формате.

Для битовых наборов в МКП реализованы следующие операции: логическое сложение, логическое умножение, неэквивалентность, отрицание, сдвиг влево, сдвиг вправо, вычисление номера старшей единицы, вычисление количества единиц, логическое умножение на логическое значение.

Логические значения являются машинным представлением значений двузначной логики и представляются в словном формате. Собственно логическое значение («истина» или «ложь») занимает нулевой разряд слова, а остальные разряды должны иметь нулевое значение. Таким образом, «ложь» представляется нулевым, а «истина» — единичным значением в нулевом разряде слова.

По форме представления логические значения являются частным случаем битовых наборов. Поэтому поразрядные операции (логическое сложение, логическое умножение, неэквивалентность, а также логическое умножение на логическое значение) распространяются и на логические значения. Однако для последних дополнительно реализованы следующие операции: логическое отрицание, выработка логического значения для битового набора, выработка логического значения для числа с плавающей точкой.

Вектор — последовательность элементов словного формата, расположенных в оперативной памяти с некоторым постоянным *шагом*, где **шаг** — это разность между адресами последовательных компонент. Вектор любого типа (целый, вещественный и др.) располагается только в одном *сегменте* и в общем случае характеризуется:

- сегментом адресного пространства;
- адресом начала вектора в этом сегменте;
- шагом;
- длиной (количеством компонент вектора).

Шаг вектора может быть положительным, отрицательным или нулевым целым числом. При нулевом шаге все компоненты вектора расположены в одной ячейке памяти.

Операции над векторами подразделяются на *покомпонентные* и *продольные*. Покомпонентные операции включают в себя унарные и бинарные операции над компонентами векторов, продольная же операция всегда выполняется над одним вектором — операндом и уменьшает длину выходного вектора до одного элемента.

Дескрипторы. В МКП используется *концепция виртуальной памяти*. Суть ее заключается в том, что адреса, к которым обращается программа во время выполнения, отделяются от адресов, реально существующих в физической памяти. Адреса, которые

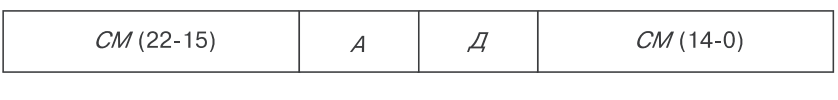
использует программа, называются *виртуальными адресами*. Адреса же, которые существуют в памяти вычислительной машины, называются *физическими адресами*. Использование виртуальной памяти освобождает программы от необходимости учета конфигурации физической памяти. Таким образом, адресное пространство программы представляет собой множество *сегментов*, расположенных в физической и виртуальной памяти. Обращение к памяти при выполнении программы сопровождается преобразованием виртуального адреса в физический, для чего используются специальные *таблицы отображения*. Эти таблицы находятся в ведении операционной системы, так как они связаны с физическим адресным пространством — ресурсом, управляемым операционной системой.

Сегменты физического и виртуального адресного пространства описываются данными адресного типа — **дескрипторами**. Дескриптор, описывающий сегмент, находящийся в физическом адресном пространстве, называется *физическим дескриптором*. Дескриптор, описывающий сегмент, расположенный в виртуальном адресном пространстве, называется *виртуальным дескриптором*. Дескрипторы (виртуальные и физические) содержат, в частности, начальный адрес и размер сегмента. Организация доступа в память с помощью дескрипторов осуществляется с использованием специальной регистровой памяти — **дисплей-регистров.** В *Д*-регистры можно заносить только виртуальные и физические дескрипторы, при этом загрузка виртуальных дескрипторов разрешена любым программам, а физических — только привилегированным процедурам ядра операционной системы. Вместе с тем, программа пользователя может сама сформировать физический дескриптор, преобразовав в него виртуальный, и использовать его для доступа в память.

Операции над дескрипторами: коррекция адреса начала, коррекция длины сегмента, преобразование виртуального дескриптора в физический.

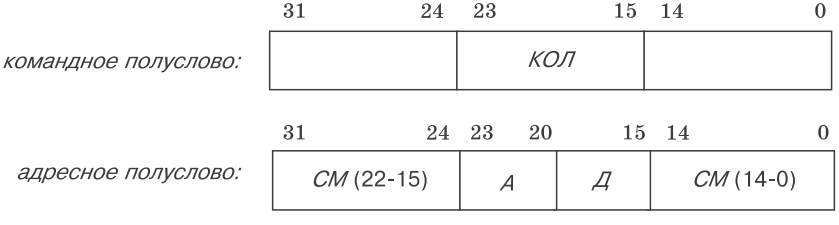
8.3. Способы адресации операндов

Схема адресации слова включает в себя номер *Д*-регистра, номер *А*-регистра и статическое смещение. Такая схема реализуется адресным полусловом:



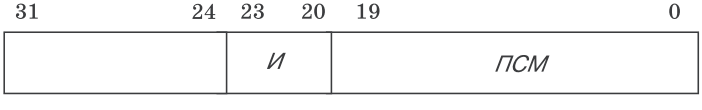
Здесь поле *Д* содержит номер *Д*-регистра, хранящего дескриптор адресуемого сегмента памяти, значение поля *А* — номер *И*-регистра, содержимое которого используется как *динамический словный индекс* (он может принимать положительные, отрицательные и нулевое значения), а неотрицательное словное смещение (*СМ*) относительно начала сегмента разделяется на две части так, что старшие разряды полуслова содержат старшие разряды значения.

Схема адресации группы слов включает в себя адресацию первого слова в сегменте памяти и указание количества последовательно расположенных слов. Такая схема реализуется двумя полусловами — командным и адресным:

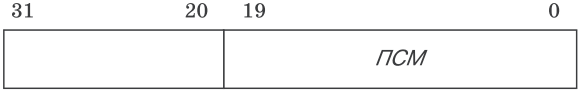


Здесь адресное полуслово определяет адрес первого слова в группе (соответствующая схема адресации была описана выше). Количество адресуемых слов в группе определяется значением, находящимся в поле *КОЛ* командного полуслова; в разных командах длина этого поля изменяется от 4 до 9 разрядов.

Адресация полуслов используется только для адресации команд (в командах перехода). Общий случай такой адресации:



Здесь в поле *И* хранится номер *И*-регистра, содержимое которого используется как динамический полусловный индекс. При этом на значение индекса накладывается ограничение: оно должно быть неотрицательным. Поле ПСМ содержит неотрицательное значение — статическое полусловное смещение относительно начала сегмента программного кода. В частном случае адресации полуслова динамический индекс отсутствует:



Блокировка индексирования. Во всех схемах динамического индексирования при задании номера 17 в поле *А*- или *И*-регистра индексирующее значение принимается равным 0 независимо от значения, находящегося в *И*17.

Адресация регистровых операндов. В командах используются две схемы задания регистровых операндов: *числовая* и *позиционная*. При числовом задании в поле операнда записывается номер регистра, а кодом операции и/или дополнительной информацией уточняется тип регистра. Если действие операции распространяется на несколько однотипных регистров с последовательными номерами, то поле операнда содержит номер первого регистра из группы. Позиционное же задание номеров регистров реализуется с использованием *шкалы* — части полной команды, в которой только для *И*- и *Д*-регистров (за исключением *Д*34–*Д*37) имеется по одному отдельному разряду для каждого регистра. Действие команды распространяется на те регистры, для которых в соответствующих разрядах шкалы записаны единицы.

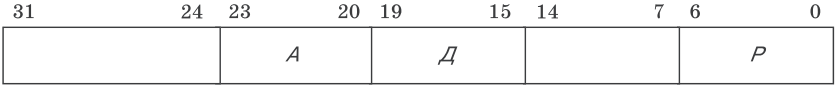
Непосредственные данные задаются в командах тремя способами:

- константными полусловами, следующими непосредственно за командным полусловом;
- коротким целым числом или коротким двоичным числом в поле операнда двухбайтовой команды;
- коротким двоичным числом, располагаемым в поле операнда полусловной команды.

Вариантное задание операнда. В некоторых командах однооперандное поле используется для задания разных источников операндов. Само операндное поле занимает младшие разряды командного полуслова, а содержимое поля определяется двухразрядным полем *ТО* (тип операнда), занимающим разряды 8 и 9 командного полуслова. Возможные двоичные значения поля *ТО* определяют следующие источники операндов:

- 00 — *Р*-регистр, номер которого находится в разрядах 0–6;
- 01 — *И*- или *А*-регистр, номер которого находится в разрядах 0–3;
- 10 — *Д*-регистр, номер которого находится в разрядах 0–4;
- 11 — короткое двоичное число, находящееся в разрядах 0–6.

Адрес вектора задается полусловом:



Здесь семантика полей *Д* и *А* такая же, как и в схеме адресации слова. Поле *Р* содержит номер *Р*-регистра, в правом полуслове которого находится неотрицательное динамическое словное смещение (второе динамическое наряду со смещением в *И*-регистре), которое участвует в формировании адреса первого компонента вектора относительно начала сегмента. Содержимое правого полуслова задается как 23-разрядное двоичное значение (старшие разряды этого полуслова игнорируются). Левое полуслово *Р*-регистра содержит значение шага вектора — 23-разрядное целое со знаком.

8.4. Команды

Рассмотрим подмножество команд МКП, которых достаточно для реализации стандарта языка Паскаль.

В соответствии с форматом представления, команды МКП делятся на группы:

- скалярные операции над числами с плавающей точкой, целыми, двоичными числами, битовыми наборами и логическими значениями;
- пересылка данных между *локальной оперативной памятью* (*ЛОП*) и регистрами;
- команды для *И*-регистров;
- команды передачи управления;
- операции управления регистровым контекстом;
- операции над дескрипторами;
- векторные операции над числами с плавающей точкой, целыми, двоичными числами, битовыми наборами и логическими значениями.

Маршруты движения данных между регистрами и локальной памятью представлены на рис. 8.1. Здесь стрелка означает наличие в системе команд операции пересылки между регистрами или локальной памятью и регистрами. Например, данные, которые находятся в ЛОП и будут передаваться на *С*-регистры,

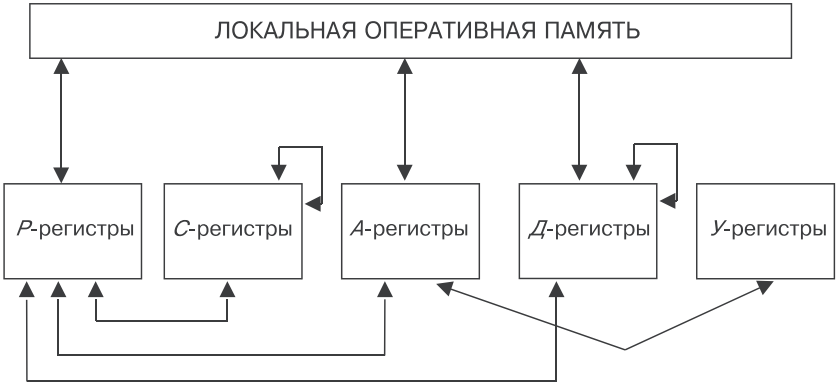


Рис. 8.1. Маршруты движения данных между регистрами и локальной памятью

сначала должны поступать в *P*-регистры, а затем — на сумматоры.

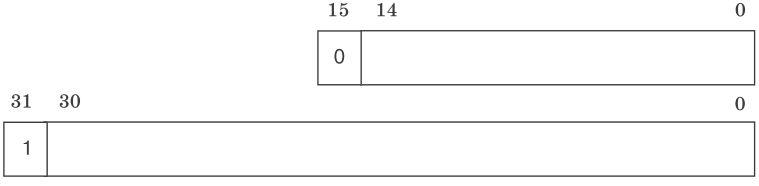
Элементы потока команд разделяются на два класса: *командные* и *операндные*. В командном элементе указываются выполняемая операция и, для большинства команд, ее операнды; в операндных элементах — только операнды операции, которая задается в одном из предшествующих, но обязательно командном элементе.

Элементы потока команд могут иметь двухбайтовую или полусловную длину. Каждый элемент двухбайтового формата является командным, а полусловный элемент может быть как командным, так и операндным.

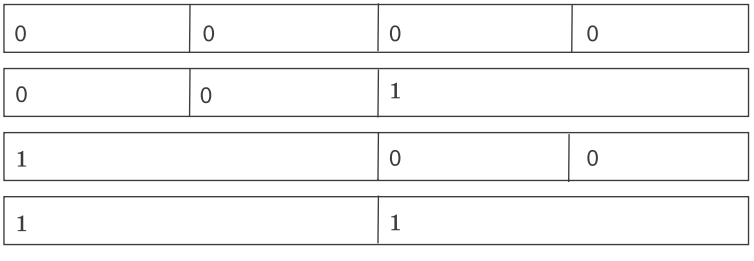
Различают *команды фиксированной* и *переменной длины*. Команды фиксированной длины состоят из элементов двухбайтового либо полусловного формата. При этом двухбайтовая команда всегда состоит из одного элемента, а полусловные команды фиксированной длины имеют два варианта структуры, где команда может состоять из одного командного полуслова или командного и операндного полуслов. Команда же переменной длины может состоять из:

- командного и нескольких операндных полуслов, количество которых фиксируется в командном полуслове;
- нескольких командных полуслов (такими командами представляются только векторные операторы).

Двухбайтовые и полусловные командные элементы имеют разное значение старшего разряда: «0» — для первых, «1» — для вторых:



Таким образом, одно слово может содержать от 2 до 4 элементов потока команд. Возможные варианты размещения элементов в слове:



В МКП осуществляется просмотр потока команд вперед и сортировка его на независимые внутренние подпотоки, выполняемые параллельно.

8.4.1. Команды для *C*- и *P*-регистров

Скалярные операции над числами с плавающей точкой, целыми, двоичными числами, битовыми наборами и логическими значениями реализуются командами, оперирующими с *P*- и *C*-регистрами. Команды для *C*- и *P*-регистров представляют из себя два вида операторов присваивания, имеющих разные правые части в зависимости от указанной в команде операции.

Если эта операция относится к числу бинарных, то оператор присваивания имеет вид:

```
Результат = Операнд1 % Операнд2,
```

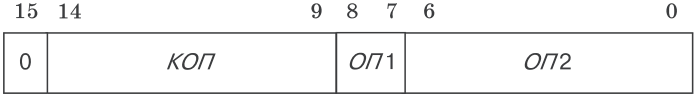
где символ % обозначает код бинарной операции.

Для унарной операции оператор присваивания записывается так:

```
Результат = % Операнд2,
```

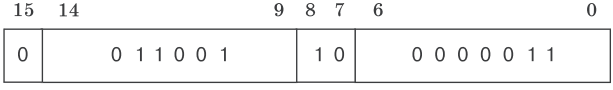
где символ % обозначает унарную операцию.

Команды для *C*- и *P*-регистров имеют двухбайтовый формат:

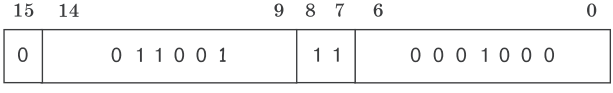


Здесь в команде имеется три поля: код операции (*КОП*), первый (*ОП1*) и второй операнд (*ОП2*), где во всех операциях поле первого операнда содержит номер *C*-регистра. Семантика поля второго операнда определяется так:

1) для бинарных и унарных операций поле *ОП2* содержит номер *C*-регистра либо номер *P*-регистра, в зависимости от находящегося в поле значения. Если это значение меньше или равно 3, то поле *ОП2* содержит номер *C*-регистра, иначе — виртуальный номер *P*-регистра. Например, битовое представление команды вычитания целых чисел имеет вид:

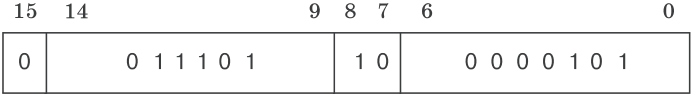


(*C* 2) <- (*C* 2) – (*C* 3)



(*C* 3) <- (*C* 3) – (*P* 10)

2) в операции пересылки константы (непосредственного данного) в *C*-регистр поле *ОП2* содержит двоичное значение пересылаемой константы. Например, битовое представление команды «Пересылка константы в *C*-регистр» имеет вид:



(*C* 2) <- 5

3) в операциях сдвига поле *ОП2* содержит короткое целое значение;

4) в операциях пересылки между *C*- и *P*-регистрами поле *ОП2* содержит виртуальный номер P-регистра (см. п. 8.4.5.2) или номер *C*-регистра.

Бинарная операция выполняется над операндами, источники которых указываются в первом и втором операндных полях команды, а унарная операция выполняется над операндом, источник которого находится во втором операндном поле (кроме пересылки «Сумматор — *P*-регистр»). Результат при этом помещается в регистр, указанный в первом операндном поле команды (таковым всегда является *C*-регистр).

8.4.2. Команды пересылки данных между локальной памятью и регистрами

Команды пересылки данных между локальной оперативной памятью и регистрами состоят из двух полуслов — командного и операндного. Формат командного полуслова:

Здесь значение поля *КОЛ*, увеличенное на 1, определяет количество регистров, участвующих в операции. Код операции (поле *КОП*) определяет следующие варианты пересылки:

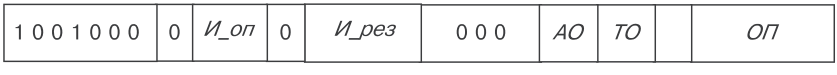
- из ЛОП в регистры (*КОП* = 01000);
- из регистров в ЛОП (*КОП* = 11000);
- непосредственного данного в регистры (*КОП* = 01010).

Поле *ОП* содержит номер первого регистра, участвующего в операции. Тип регистра задается полем *ТО* (см. п. 8.3). При этом номер последнего регистра из группы не должен превышать максимальный номер регистра соответствующего типа. Операндное полуслово содержит адрес локальной памяти или непосредственное данное; оно размножается необходимое количество раз, и операция выполняется над последовательными парами «регистр — размноженный операнд». Размножение константного полуслова производится простым дублированием его значения, а размножение адресного полуслова — последовательным увеличением значения в поле смещения на единицу.

8.4.3. Команды для *И*-регистров

8.4.3.1. Скалярные операции над целыми числами, формирующие результат в *И*-регистрах (индексные команды)

Индексными командами реализуются операции: сложение, вычитание, вычитание обратное. Формат индексной команды:

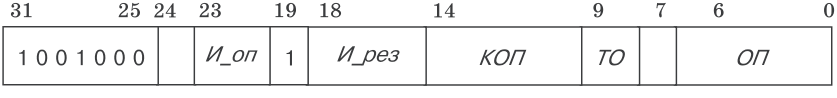


Первый операнд находится в *И*-регистре, определяемом полем *И_оп*. На этот операнд распространяется правило *блокиров-*

ки индексирования: если значение в поле *И_оп* равно 17, то в качестве первого операнда выбирается целочисленный нуль. Второй операнд может выбираться из *И*-, *Р*-регистра или являться непосредственным данным. Поле *ТО* может принимать одно из следующих значений: 00, 01, 11 (см. п. 8.3). Результат операции помещается в *И*-регистр, номер которого находится в поле *И_рез*. Поле же *АО* содержит код операции: $АО = 0$ — сложение, $АО = 1$ — вычитание, $АО = 3$ — обратное вычитание.

8.4.3.2. Пересылки данных между *И*- и *Р*-регистрами

Формат команд пересылки:



Здесь значение поля *КОП* определяет два варианта пересылок, а именно:

1) $КОП = 0$:
- $ТО = 00$ — *Р*-регистр → *И*-регистр; номер *Р*-регистра задается в поле *ОП*, а номер *И*-регистра — в поле *И_рез*;
- $ТО = 01$ — *И*-регистр → *И*-регистр; значение из регистра *И_оп* пересылается в регистр *И_рез*;

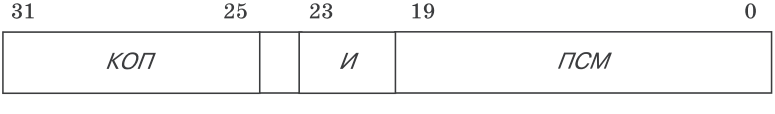
2) $КОП = 1$, $ТО = 00$ — *И*-регистр → *Р*-регистр; значение из *И*-регистра, номер которого находится в поле *И_оп*, пересылается в *Р*-регистр, указываемый полями *ТО* и *ОП*.

8.4.4. Команды передачи управления

Команды передачи управления делятся на две группы:
- внутримодульные передачи управления;
- межмодульные передачи управления.

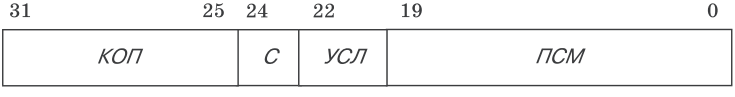
Команды внутримодульных передач управления также распределяются на три группы:
- безусловные переходы;
- условные переходы по значению *С*-регистра;
- условные переходы по значению *И*-регистра.

Команды безусловного перехода «Переход» и «Переход с сохранением адреса возврата» имеют следующий формат ($КОП =$ 1010011 и 1001101 соответственно):



При этом поле относительного адреса перехода (*ПСМ*) содержит значение полусловного смещения в сегменте программного кода. Обе команды имеют два варианта выполнения — с индексированием и без индексирования адреса перехода. Значение 17 в поле *И*-регистра определяет вариант выполнения команды без индексирования адреса перехода.

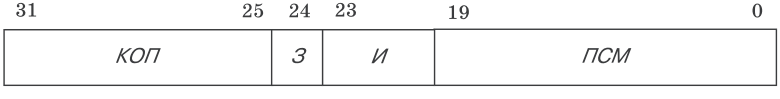
Переходы по значению С-регистра реализуют передачи управления, зависящие от содержимого *С*-регистра. Номер *С*-регистра и код условия перехода (*УСЛ*) при этом указывается в соответствующих полях команды:



Некоторые коды условий для передач управления:

- *УСЛ* = 0 — переход по значению «ложь» в *С*-регистре;
- *УСЛ* = 1 — переход по значению «истина» в *С*-регистре;
- *УСЛ* = 6 — переход, если в *С*-регистре находится неотрицательное значение;
- *УСЛ* = 7 — переход, если в *С*-регистре находится отрицательное целое число.

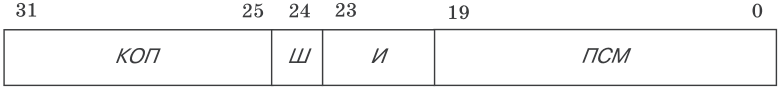
Переходы по значению И-регистра реализуют передачи управления, зависящие от содержимого *И*-регистра. Номер *И*-регистра указывается в поле *И* командного полуслова:



Код операции (*КОП*) вместе с признаком *З* конкретизирует следующие условия перехода:

- *КОП* = 125, *З* = 0 — целое значение не равно нулю;
- *КОП* = 125, *З* = 1 — целое значение равно нулю;
- *КОП* = 126, *З* = 0 — целое значение больше или равно нулю;
- *КОП* = 126, *З* = 1 — целое значение меньше нуля;
- *КОП* = 127, *З* = 0 — целое значение меньше или равно нулю;
- *КОП* = 127, *З* = 1 — целое значение больше нуля.

Переходы по концу цикла реализуют передачи управления, автоматически изменяя значения *И*-регистров, указанных в команде. Формат команды перехода по концу цикла:

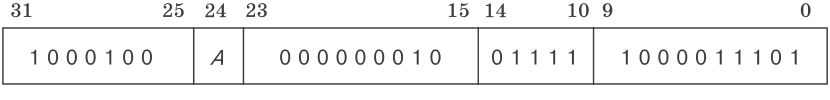


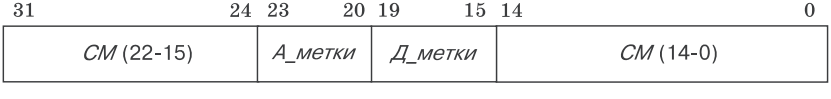
Здесь с командой связано два регистра. Первый (с номером *И*) содержит текущее значение параметра цикла, второй (с номером *И* + 1) — конечное значение параметра цикла. Шаг изменения параметра цикла является константой, равной 1 или −1 в зависимости от значения поля *Ш* командного полуслова (1 или 0 соответственно). Этот вариант перехода по концу цикла реализуется командами, имеющими значение *КОП*, равное 121. Выполнение команды «Конец цикла» состоит в пересчете значения параметра цикла и, если цикл не завершен, в переходе по адресу, указанному в поле *ПСМ* (см. табл. 8.1).

Таблица 8.1

Шаг изменения параметра цикла	Условие завершения цикла
1	пересчитанное значение больше конечного значения
−1	пересчитанное значение меньше конечного значения

Межмодульная передача управления реализуется командой «Межмодульный переход», формат которой имеет вид:





Здесь команда состоит из двух полуслов — командного и адресного, где адресное полуслово определяет виртуальный адрес метки входа в подпрограмму, на которую осуществляется переход.

8.4.5. Управление регистровым контекстом

Операции, операнды которых находятся в регистровой памяти, выполняются быстрее, чем такие же операции над операндами оперативной памяти. Поэтому в процессе генерации кода регистры активно используются для управления циклами, адресации переменных, хранения промежуточных результатов вычисления выражений и в иных целях.

Основная программа, а также каждая процедура (функция) имеют свой регистровый контекст. Поэтому в момент вызова

процедуры (функции) необходимо сохранять содержимое регистров, а перед возвратом управления вызывающей процедуре (функции) — восстанавливать содержимое регистров.

Для переключения регистрового контекста во время вызова/завершения процедур (функций) в МКП используется *стековый механизм управления регистровой памятью*.

8.4.5.1. Управление *А*-, *Д*- и *У*-регистрами

Сегмент локальной памяти, дескриптор которого находится в регистре *Д*36, используется для сохранения и восстановления значений *А*-регистров (*А*00–*А*17), *Д*-регистров (*Д*00–*Д*35) и некоторых *У*-регистров. Этот сегмент памяти имеет стековую организацию и называется **аппаратурным стеком.**

При сохранении содержимого регистров в аппаратурный стек выполняются следующие действия:

- фиксируется шкала сохраняемых регистров;
- формируется очередная зона сохранения, соответствующая шкале.

В шкале сохраняемых регистров каждый разряд «связан» с одним или несколькими регистрами. Единичное значение разряда шкалы свидетельствует о том, что содержимое соответствующего регистра (или регистров) должно быть записано в зоне сохранения. Значения регистров, которым в шкале соответствуют разряды с нулевым значением, в стек не сохраняются. Разряды шкалы с 10 по 39 и с 40 по 55 соответствуют *Д*-регистрам *Д*35–*Д*00 и *А*-регистрам *А*17–А00 (именно в указанном порядке!). *Д*-регистры *Д*36 и *Д*37 в стек не сохраняются и из стека не восстанавливаются.

В вершине зоны сохранения находится шкала сохраненных регистров, а в располагаемых ниже словах — значения регистров (рис. 8.2).

При этом 63-й разряд шкалы соответствует группе управляющих регистров — *АКОМ* и др. (*АКОМ* — это сокращенное название *У*-регистра, в котором находится адрес очередной выбираемой из памяти команды). 62-й разряд шкалы соответствует *У*-регистру, называемому «маской сохраняемых регистров». 61-й разряд шкалы соответствует группе управляющих регистров: *РБР*, *РГР* и др., значения которых совместно сохраняются в одном слове зоны сохранения (см. п. 8.4.5.2).

Все операции над аппаратурным стеком разбиваются на две группы. Операции первой группы сохраняют значения регистров в стековый сегмент, а второй — восстанавливают значения регистров из стека.

63 … 0	*Соответствующие разряды шкалы:*
Шкала сохраненных регистров	
*Д*35	10
*Д*34	11
. . .	
*Д*00	39
*А*17	40
. . .	
*А*00	
РБР, *РГР*, . . .	61
РМС	62
АКОМ, . . .	63

Рис. 8.2. Структура полной зоны сохранения

Рассмотрим два варианта фиксации шкалы сохраняемых регистров: аппаратный и программно-аппаратный.

В аппаратном варианте используется стандартная шкала. Команды «Переход с сохранением адреса возврата» и «Межмодульный переход» сохраняют в стек информацию, определяемую стандартной шкалой (рис. 8.3):

Рис. 8.3. Варианты шкал сохранения в командах:
а) «Переход с сохранением адреса возврата»,
б) «Межмодульный переход»

В программно-аппаратном варианте фиксации шкалы, который имеет место при выполнении команды «Начало блока», используется комбинация стандартного и программного способов формирования шкалы. При этом шкала стандартно получает единицу в 61-м и 62-м разрядах:

Поля шкалы, соответствующие упрятываемым *А*- и *Д*-регистрам, задаются в команде «Начало блока»:

Для восстановления состояния регистров из «верхней» зоны сохранения извлекается шкала сохраненных регистров. Затем производится восстановление содержимого регистров, определяемых шкалой. Операции восстановления реализуются командами «Конец блока» и «Возврат».

По команде «Конец блока» («Возврат») из стека последовательно извлекаются зоны сохранения и распределяются по регистрам. Команда завершается при поступлении зоны, содержащей «1» в 62 (63)-м разряде шкалы.

В системе команд МКП существует две группы команд, каждая из которых включает в себя взаимно соответствующее сохранение и восстановление. В первую группу входят операции «Переход с сохранением адреса возврата», «Межмодульный переход» и «Возврат», которые обеспечивают передачу управления на программный код вызываемой процедуры и возврат в вызывающую процедуру. Вторую группу составляют команды «Начало блока» и «Конец блока», сохраняющие и восстанавливающие состояния *А*-, *Д*- и *У*-регистров.

Использование команд «Переход с сохранением адреса возврата», «Начало блока» и «Возврат» показано на рис. 8.4.

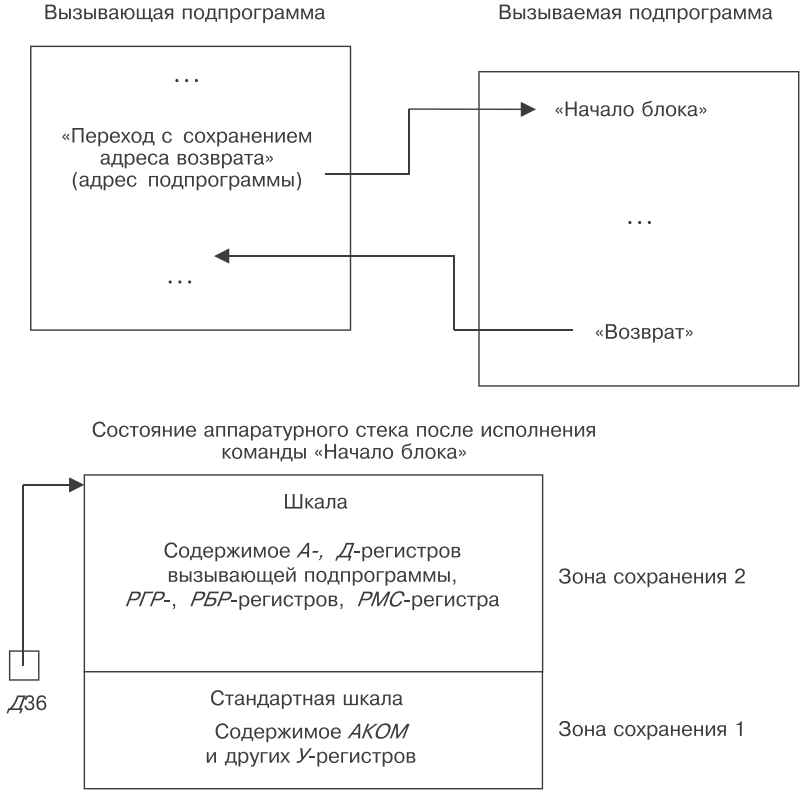


Рис. 8.4. Использование команд «Переход с сохранением адреса возврата», «Начало блока» и «Возврат»

8.4.5.2. Управление *Р*-регистровой памятью

Р-регистровая память предназначена в основном для хранения локальных данных программы, а также процедур (функций). Схема адресации *Р*-регистра включает в себя указание его номера. В МКП используется виртуальная адресация *Р*-регистров. Программные номера 4–127 всегда определяют виртуальные номера *Р*-регистров. В командах обращения к *Р*-регистровой памяти виртуальные номера *Р*-регистров динамически преобразуются в физические.

Для управления *Р*-регистровой памятью имеется два *У*-регистра: *Регистр базы* и *Регистр границ* (сокращенно — *РБР* и *РГР*). Здесь регистр *РБР* содержит адрес начала сегмента физических *Р*-регистров, где номер физического *Р*-регистра равен

сумме значения регистра *РБР* и виртуального номера. Такое преобразование виртуального номера в физический (с учетом диапазона виртуальных номеров *Р*-регистров) оставляет недоступными для программы *Р*-регистры с номерами 0–3 относительно текущего значения *РБР*. При адресации *Р*-регистров номера 0–3 всегда определяют соответствующие физические регистры, которые, таким образом, имеют глобальный характер. Регистр же *РГР* хранит длину «зоны обработки», т. е. максимальный виртуальный номер используемого в программе (процедуре или функции) *Р*-регистра (рис. 8.5).

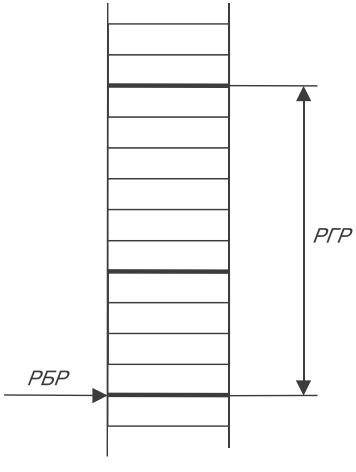


Рис. 8.5. Зона обработки *Р*-регистров

В программе должны быть указаны виртуальные номера *Р*-регистров, не превышающие находящегося в *РГР* значения. Контроль возможного превышения в динамике работы не производится, так как предполагается, что любая программа (процедура или функция) устанавливает ту зону обработки, которая ей необходима. Пустой зоне обработки соответствует значение *РГР*, равное 3. (На рис. 8.6 недоступные *Р*-регистры закрашены серым цветом.)

Таким образом, пара регистров *РБР* и *РГР* определяет *Р*-регистровую зону обработки программы (процедуры или функции).

Команды, реализующие вызов процедур (функций), сохраняют содержимое регистров *РБР* и *РГР* в сегмент локальной памяти, адресуемый *Д*-регистром *Д*36. После возврата из процедуры (функции) старый *Р*-регистровый контекст должен

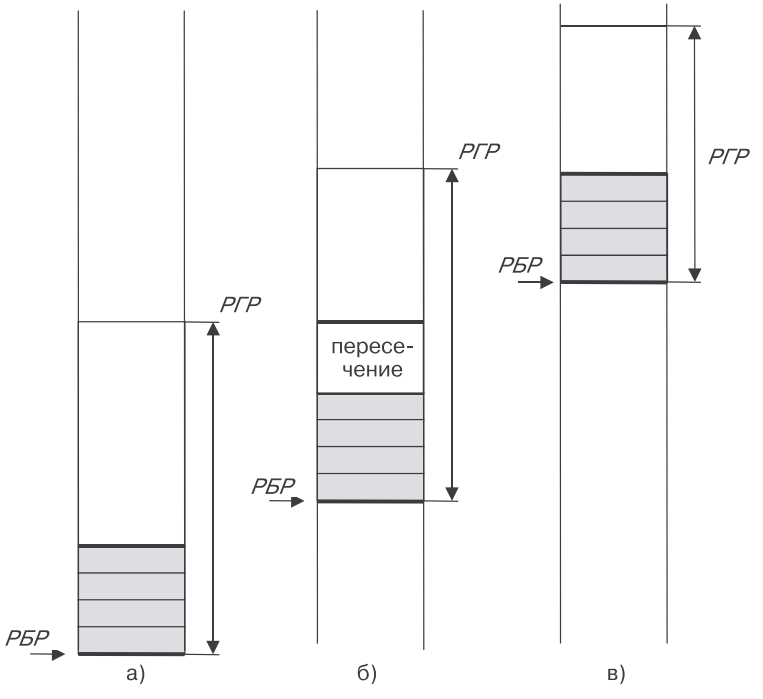


Рис. 8.6. Сдвиг зоны обработки с одновременным изменением длины: а) — исходная зона обработки, б) — сдвиг в сторону старших номеров *Р*-регистров при входе в процедуру (новая зона обработки пересекается со старой), в) — сдвиг в сторону старших номеров *Р*-регистров при входе в процедуру (новая зона устанавливается непосредственно за старой)

быть восстановлен. Управление *Р*-регистровой памятью производится программно. Система команд предоставляет возможность динамически изменять местоположение зоны обработки и ее длину. Команды управления зоной обработки приведены на рис. 8.7–8.9.

Операцией сдвига обеспечивается перемещение зоны только в сторону старших номеров физических *Р*-регистров. При этом

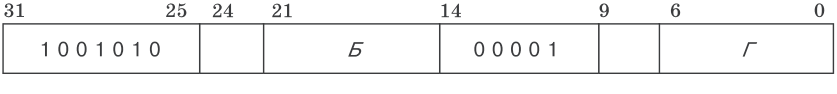


Рис. 8.7. Формат команды установки начала и длины зоны обработки. Действие этой команды: значения из полей *Б* и *Г* пересылаются в регистры *РБР* и *РГР* соответственно

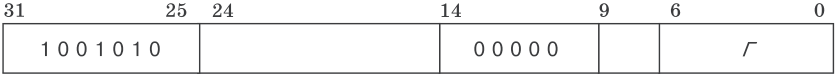


Рис. 8.8. Формат команды изменения длины зоны обработки с сохранением ее начала. Действие этой команды: значение из поля *Г* пересылается в регистр *РГР*

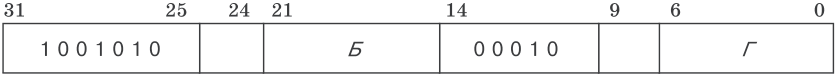


Рис. 8.9. Формат команды сдвига зоны обработки с одновременным изменением ее длины. Действие этой команды: значения регистров *РБР* и *РГР* изменяются по формулам *РБР* = *РБР* + *РГР* − *Б* и *РГР* = *Г*

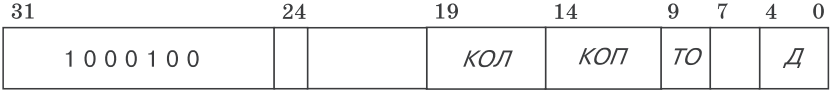
новая зона обработки может пересекаться со старой либо устанавливаться непосредственно за старой зоной. Пересечение зон обработки вызывающей и вызываемой процедур может быть использовано для передачи параметров. Однако такой сдвиг в сторону старших номеров, при котором между старой и новой зонами возник бы «просвет» из регистров, невозможен. Команда «Конец блока» восстанавливает старые значения *РБР* и *РГР* и, следовательно, обеспечивает сдвиг зоны обработки в сторону младших номеров физических *Р*-регистров.

8.4.6. Команды для *Д*-регистров

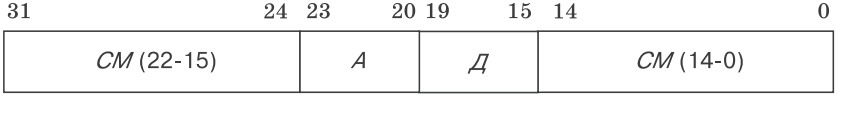
8.4.6.1. Операции над дескрипторами

Операции над дескрипторами — это команды переменной длины, состоящие из командного и адресных полуслов. Формат операций над дескрипторами:

командное полуслово:



адресное полуслово:



Действие команды «Коррекция длины сегмента» (*КОП* = 05, *ТО* = 10) заключается в следующем: в дескрипторах, выбираемых из определяемых адресными полусловами *Д*-регистров, устанавливаются поля длины, равные значениям индексированных смещений из соответствующих адресных полуслов. После этого измененные дескрипторы помещаются в *Д*-регистры, начиная с номера *Д*, указанного в командном полуслове.

Действие команды «Коррекция адреса начала» (*КОП* = 06, *ТО* = 10): в дескрипторах, выбираемых из определяемых адресными полусловами *Д*-регистров, увеличиваются значения в полях начала и уменьшаются значения в полях длины на величины индексированных смещений соответствующих адресных полуслов, после чего измененные дескрипторы помещаются в *Д*-регистры, начиная с номера *Д*.

8.4.6.2. Пересылки дескрипторов между *Д*- и *Р*-регистрами

Формат команд пересылок между *Д*- и *Р*-регистрами:

общий формат:

31 … 24	19 … 14	9 … 7	6 … 0
1 0 0 1 0 0 1	*Оп/Рез*	*КОП* *ТО*	*ОП*

пересылка ОПеранд – Д-регистр:

31 … 24	19 … 14	9 … 7	6 … 0
1 0 0 1 0 0 1	*Дрез*	00000 *ТО*	*ОП*

пересылка Д-регистр – Р-регистр:

31 … 24	19 … 14	9 … 7	6 … 0
1 0 0 1 0 0 1	*Доп*	00000 *ТО*	00 *Р*

пересылка Д-регистр – Д-регистр с преобразованием дескриптора:

31 … 24	19 … 14	9 … 7	6 … 0
1 0 0 1 0 0 1	*Дрез*	00000 *ТО*	10 *Доп*

Команда содержит поле основного кода команды (111), кода операции (*КОП*), *Д*-регистра — операнда или результата (*Оп/Рез*) и вариантного операнда (*ОП*). Источники вариантного операнда задаются так:

- *ТО* = 0 — источник — *Р*-регистр;
- *ТО* = 2 — источник — *Д*-регистр;
- *ТО* = 1 или 3 — запрещенные источники операндов.

Значение поля *КОП* определяет следующие варианты пересылок:

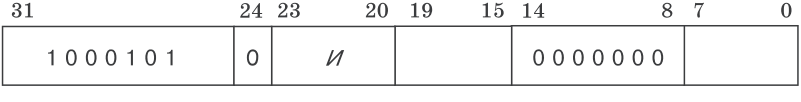
- *КОП* = 0 — пересылка дескриптора из регистра, определяемого полями *ТО* и *ОП*, в *Дрез*;
- *КОП* = 1 — пересылка дескриптора из *Доп* в *Р*-регистр; в этой операции всегда требуется *ТО* = 0, другие варианты задания операнда не допускаются;
- *КОП* = 2 — пересылка из *Доп* в *Дрез* с преобразованием виртуального дескриптора в физический.

8.4.7. Векторные команды

Для реализации стандарта языка Паскаль нам также потребуется операция «пересылка из вектора в вектор».

Все действия над векторами оформляются в виде **векторного оператора**. Это команда переменной длины, состоящая из нескольких *векторных команд*, последовательное расположение которых подчиняется определенным правилам, а векторные команды представляются командными полусловами.

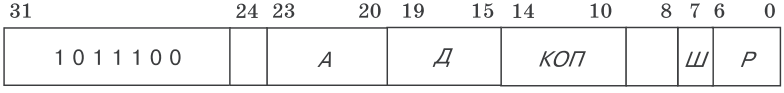
Векторный оператор начинается командой «Начало векторного оператора», формат которой имеет вид:



Здесь неотрицательное целое значение, находящееся в *И*-регистре, фиксирует единую длину обрабатываемых векторов.

Команда «Пересылка из вектора в вектор» использует два типа векторов: чтения и записи. *Вектор чтения* — это исходный вектор, а *вектор записи* — результирующий вектор. В векторном операторе вектор чтения, входной вектор последующей унарной операции и вектор записи определяются отдельными командами.

Формат команды «Вектор в локальной памяти»:



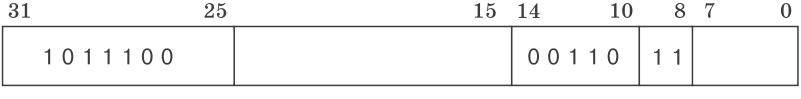
Здесь поля *КОП* и *Ш* содержат следующие значения для определяемых командой векторов:

- *КОП* = 00, *Ш* = 0 — вектор чтения;
- *КОП* = 20, *Ш* = 0 — вектор записи результата предыдущей покомпонентной операции.

Поля же *А*, *Д* и *Р* задают адрес вектора.

Первая команда определения вектора записи в локальной памяти завершает векторный оператор.

Формат команды «Пересылка из вектора в вектор»:



Таким образом, векторный оператор, реализующий пересылку из вектора в вектор, состоит из следующей последовательности векторных команд:

1) начало векторного оператора;

2) вектор в локальной памяти (определение вектора чтения);

3) унарная операция «Пересылка из вектора в вектор»;

4) вектор в локальной памяти (определение вектора записи).

Глава 9

Организация оперативной памяти во время выполнения программы

9.1. Области данных процедур

Язык Паскаль реализует блочную структуру программы, которая требует управления памятью в процессе ее выполнения. С каждым вызовом процедуры (функции) связывается динамическая область данных фиксированной длины, в которой хранятся:

- значения фактических параметров, передаваемых по значению;
- адреса фактических параметров, передаваемых переменной;
- локальные переменные;
- административная информация.

Когда процедура вызвана, она забирает под свою область данных необходимый объем памяти. Когда выполняется выход из процедуры, память, занятая областью данных, освобождается. При этом первой всегда заканчивается процедура, выполнение которой началось позже других, поэтому память под области данных организована в виде стека.

Рассмотрим организацию *стека областей данных* для программы с вложенными описаниями процедур:

```
program example;
   var prog1, prog2, prog3: integer;
    ...
   procedure A ;
      var a1,a2,a3: integer;
     procedure B ;
      var b1,b2: real;
     begin  b2:= prog1*a1; ... end;
     procedure C ;
       var c1,c2: real;
     begin   ...  B ;   ...    end;
    begin { A }  ...  C ;   ...    end;
 begin { основная программа }
   ...   A ; ...
end.
```

Здесь сначала выделяется память для области данных основной программы. В момент вызова процедуры **A** выделяется память под область данных этой процедуры. Далее вызывается процедура **C**. Тогда к моменту вызова процедуры **B** стек принимает вид, изображенный на рис. 9.1.

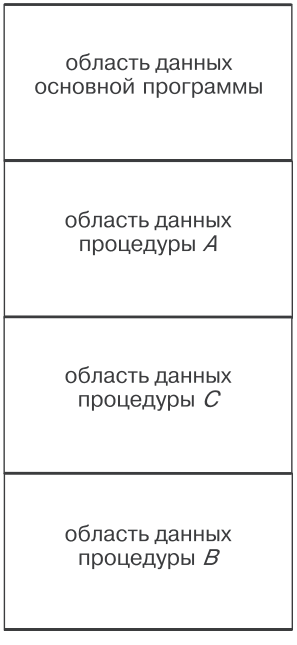


Рис. 9.1. Стек областей данных

Если процедура вызывается рекурсивно, то для этой процедуры создается несколько областей данных, каждая — для отдельного ее вызова.

9.2. Адресация переменных

Во время выполнения программы возможны обращения к переменным из нескольких областей данных. Если последней активной является процедура уровня I, то она может обращаться кроме своих собственных к переменным процедур уровней 1, 2, ..., $I - 1$. В нашем примере в момент выполнения процедуры `B` (уровень 3) доступны переменные из областей данных основной программы (уровень 1), процедуры **A** (уровень 2) и,

конечно же, собственные переменные процедуры **B**. В процедуре же **C** доступ к переменным процедуры **B** невозможен, так как они имеют один и тот же уровень вложенности. Важно организовать работу с областями данных так, чтобы обращения к переменным из разных областей действия (основной программы, процедуры, функции) выполнялись правильно.

9.2.1. Адресация простых переменных

Адрес переменной во время выполнения программы вычисляется по следующей формуле:

адрес области данных + смещение переменной в этой области данных

Для вычисления адреса переменной необходимо знать:

- уровень вложенности области действия, в которой описана переменная;
- смещение переменной в области данных.

Эта информация накапливается в таблице идентификаторов во время анализа описаний переменных:

```
struct treenode
 {
  unsigned hashvalue  /* значение
                                    hash - функции */;
  char *idname       /* адрес имени в таблице имен */;
  unsigned class          /* способ использования */;
  TYPEREC  *idtype;                    /* указатель на
                                   дескриптор типа */
  union
  { /* для констант */
     union const_val constvalue;          /* значение */
     /* для процедур ( функций ) */
     struct
        { struct idparam  *param          /* указатель
                        на информацию о параметрах */;
          int forw                      /* информация об
                                опережающем описании */;
        } proc;
     /* для переменных */
     struct
```

```
        { unsigned staticlevel              /* уровень
                                  вложенности переменной */,
          offset                      /* смещение переменной
                                        в области данных */;
        } vars;
    } casenode;
  struct treenode *leftlink ;
  struct treenode *rightlink;
};
```

Введем внешнюю переменную

unsigned level

которая содержит текущий уровень вложенности процедуры (функции). Для основной программы **level** = 1. Значение переменной **level** увеличивается на 1, когда начинается анализ описания процедуры (функции), и уменьшается на 1 по завершении анализа процедуры (функции). Поле **staticlevel** для всех переменных текущей области действия принимает значение **level**.

Чтобы вычислить смещения локальных переменных и параметров, с каждым описанием процедуры (функции) необходимо связать переменную

unsigned nextlocal

содержащую смещение первой свободной ячейки памяти в области данных. Будем считать, что начало области данных (ячейки с 0 по $N-1$) занято служебной информацией. Поэтому, как только начинается трансляция процедуры (функции), переменной **nextlocal** присваивается значение N. После анализа типа переменной к значению **nextlocal** добавляется величина, равная объему памяти, который необходим для хранения значения этой переменной. По окончании обработки описаний параметров и локальных переменных **nextlocal** содержит размер области данных. Полученное значение переменной **nextlocal** запоминается в элементе таблицы идентификаторов, содержащем информацию об анализируемой процедуре (функции), и в дальнейшем используется для выделения памяти под область данных процедуры (функции). В связи с этим еще раз уточним описание структуры **treenode**:

```
struct treenode
 {
  unsigned hashvalue  /* значение
                                    hash-функции */;
  char *idname       /* адрес имени в таблице имен */;
  unsigned class          /* способ использования */;
  TYPEREC *idtype;  /* указатель на дескриптор типа */
  union
   { /* для констант */
     union const_val constvalue;          /* значение */
     /* для процедур ( функций ) */
     struct
       { struct idparam *param            /*указатель
                        на информацию о параметрах */;
         int forw                       /* информация об
                              опережающем описании */;
         int count_locals             /*размер области
                         данных процедуры (функции) */
       } proc;
     /* для переменных */
     struct
       { unsigned  staticlevel              /* уровень
                             вложенности переменной */,
         offset                  /* смещение переменной
                                   в области данных */;
       } vars;
   } casenode;
  struct treenode *leftlink ;
  struct treenode *rightlink;
 };
```

Продолжим рассмотрение нашего примера. Для машин с небольшим количеством регистров стек областей данных (см. рис. 9.1) может иметь вид, показанный на рис. 9.2.

Здесь статические указатели содержат информацию о вложенности процедур по описаниям и используются для вычисления адреса переменной во время выполнения программы. Динамические же указатели определяют вложенность процедур по возврату управления.

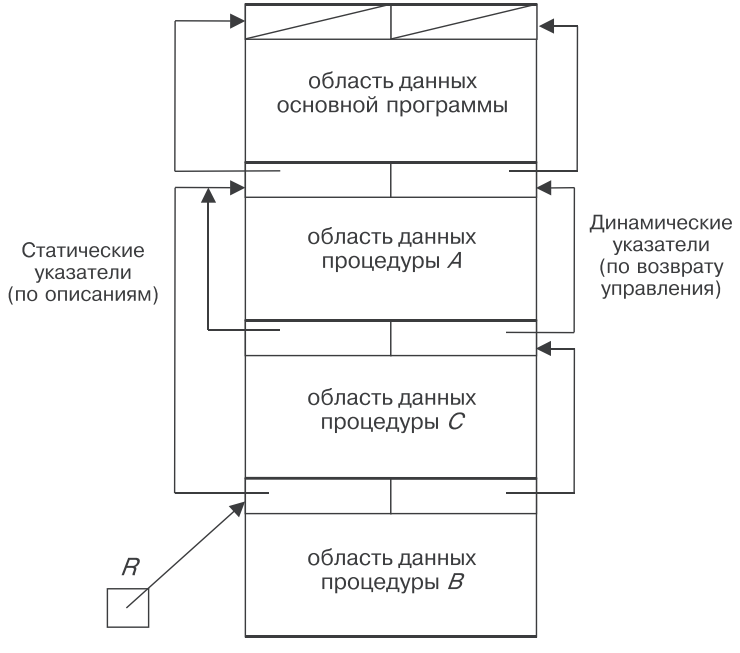


Рис. 9.2. Организация областей данных для компьютеров с традиционной архитектурой

Для хранения адреса последней активной области данных выделяется отдельный регистр. Обозначим его как **R**.

Если в процедуре **B** встречается обращение к локальной переменной, например **b2**, то ее адрес во время выполнения вычисляется как **(*R*) + *смещение в области данных***.

Теперь рассмотрим использование статической цепочки для вычисления адреса нелокальной переменной **prog1** (см. листинг в начале главы) во время выполнения операторов процедуры **B**. Обрабатывая прикладное вхождение переменной, анализатор:

1) ищет идентификатор **prog1** в таблице идентификаторов;
2) извлекает значение уровня вложенности переменной **prog1** и ее смещение в области данных из найденной вершины таблицы.

Уровень вложенности переменной **prog1** равен 1. В этот момент уровень вложенности последней активной области данных равен 3. Это означает, что генератор кода должен сгенериро-

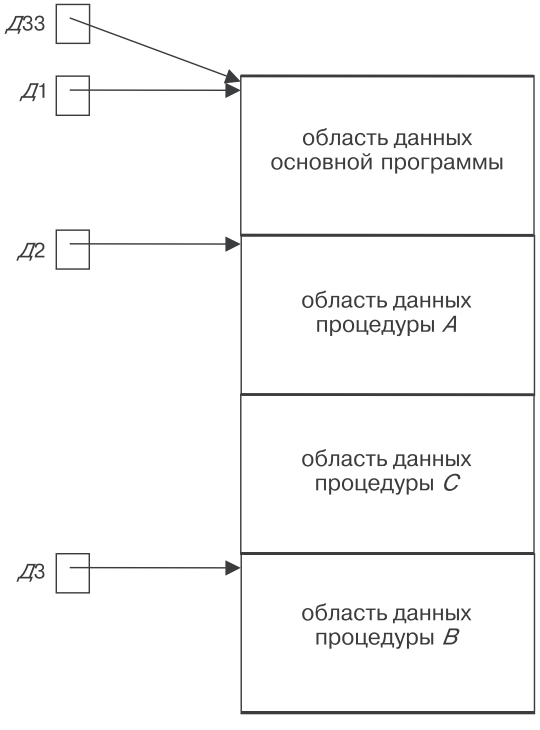


Рис. 9.3. Организация областей данных для компьютеров с достаточно большим количеством регистров

вать команды, реализующие 2 (3–1) подъема по статической цепочке: сначала от области данных **B** — к области данных процедуры **A**, а затем от области данных **A** — к области данных основной программы. Таким образом, будет найден адрес области данных, в которой располагается переменная **prog1**. В результате суммирования этого адреса со смещением переменной **prog1** мы получим абсолютный адрес этой переменной.

Архитектура МКП предоставляет возможность хранить адреса областей данных на *Д*-регистрах, а не в локальной оперативной памяти. В этом случае существенно упрощается процедура вычисления адреса переменной, так как:

- работа со статическими указателями исключается;
- динамические указатели не используются, потому что адрес области данных вызывающей процедуры запоминается в аппаратурном стеке.

Для хранения областей данных процедур (функций) удобно воспользоваться *программным стеком*, адрес которого содержится в регистре *Д*33. Физические дескрипторы сегментов областей данных процедур (функций) хранятся на *Д*-регистрах, номера которых определяются в соответствии с уровнем вложенности процедур (функций). Дескриптор области данных основной программы хранится на регистре *Д*1, так как ее уровень вложенности равен 1. Тогда стек областей данных в момент вызова процедуры **B** будет выглядеть так, как показано на рис. 9.3. Чтобы найти адрес переменной во время выполнения программы, достаточно воспользоваться формулой:

(ДI)+ смещение переменной в области данных

где ***I*** — уровень вложенности переменной.

9.2.2. Адресация переменных с индексами

Пусть объявлен массив:

```
var a: array [l1..u1, l2..u2, ... , ln..un] of type1;
```

Элементы массивов обычно размещаются в последовательных ячейках линейной одномерной непрерывной памяти. Вопрос в том, как обратиться к элементу **a[i,j,k,..,l,m]** во время выполнения программы. Так как в общем случае значения выражений **i**, **j**, **k**, ... **l**, **m** становятся известны только во время выполнения программы, то для нахождения адреса переменной необходимо генерировать машинный код. Однако сначала нужно определить формулу, в соответствии с которой будет определяться местоположение индексированной переменной.

Будем считать, что массив хранится в памяти по строкам.

```
адрес(a[i,j,k,...,l,m])=адрес(a[l1,l2,...,ln])+
                         number*size_words(type1)
```

где **number** — порядковый номер элемента **a[i,j,k,...,l,m]** относительно начала массива минус единица, а функция **size_words (type1)** возвращает размер значения типа **type1** в минимально адресуемых единицах памяти.

Введем обозначение:

```
d1 = u1 - l1 + 1; d2 = u2 - l2 + 1; ... dn = un - ln + 1;
```

т. е. **di** — количество различных значений индексов в *i*-м измерении.

Рассмотрим несколько случаев.

1) `n = 1`, т. е.

```
a: array [l1..u1]of type1;
```

Тогда

```
адрес(a[i])=адрес(a[l1])+(i-l1)*size_words(type1);
```

2) `n = 2`, т. е.

```
a: array [l1..u1, l2..u2] of type1;
```

Тогда

```
адрес (a[i,j])=адрес(a[l1,l2])+((i-l1)*d2(j-l2))*
                                size_words(type1);
```

3) `n = 3`, т. е.

```
a: array[l1..u1,l2..u2,l3..u3] of type1;
```

Тогда

```
адрес(a[i,j,k]) =
адрес(a[l1,l2,l3])+(((i-l1)*d2+(j-l2))*d3+
                    (k-l3))*size_words(type1).
```

В общем случае (можно доказать по индукции):

```
адрес(a[i,j,k,...,l,m])=
   адрес(a[l1,l2,...,ln])+(...((i-l1)*d2
                +(j-l2))*d3+ ... )*dn+(m-ln).
```

Эта формула используется для генерации кода вычисления адреса индексированной переменной.

9.2.3. Адресация поля записи

Адрес поля записи равен сумме адреса переменной — записи и смещения поля относительно начала записи.

9.3. Память для данных скалярных типов

Типы данных исходной программы должны быть отображены в форматы представления данных машины:

- целые числа и символьные значения хранятся в виде стандартных целых чисел;
- значения вещественного типа представляются как числа с плавающей точкой;
- булевы значения отображаются на машинное представление логических значений;
- указатели хранятся как битовые наборы.

Стандартное представление типа, являющегося перечислением, заключается в отображении значений в заданном порядке на машинные целые числа от 0 до $N-1$, где N — мощность типа.

При отображении отрезка в формат данных машины каждое значение представляется так же, как оно представлялось бы в своем базовом типе. В случае преобразования значения базового типа в тип отрезка производится проверка, лежит ли значение в указанном диапазоне. Проверка отношения порядка выполняется с помощью машинных команд над целыми числами. Стандартные функции **`succ(x)`** и **`pred(x)`** содержат операции сложения и вычитания единицы и проверки принадлежности результата заданной области.

9.4. Память для данных структурных типов

Одним из способов представления значения типа запись (без вариантной части) является занесение значений компонент в последовательные ячейки в порядке, указанном в описании типа. В представлении без упаковки под каждое значение компоненты отводится целое количество слов. Если же значения таковы, что их можно уместить в памяти, объем которой меньше одного слова, то появляется возможность упаковки в слове нескольких компонент. В представлении с упаковкой последовательности битов для различных компонент располагаются друг за другом.

При отображении вариантной части записи необходимо представить поле признака. Значения различных вариантов требуют для хранения разные объемы памяти. Если используется представление без упаковки, то поле признака помещается в отдельное слово, поэтому количество памяти, занимаемое вариантной частью, на одно слово превышает память, необходимую для хранения самого большого варианта.

Выбирая машинное представление для множеств, желательно сделать так, чтобы каждая из основных операций выполнялась одной машинной командой, а объем необходимой памяти был по возможности минимален. Для множеств эти требования могут быть полностью удовлетворены при условии, что базовый тип не слишком велик. Способ их представления при этом заключается в том, чтобы отводить для множества в памяти столько битов, сколько в множестве потенциальных элементов. Таким образом, каждому значению базового типа соответствует индивидуальный бит, принимающий значение «1», если значение является элементом множества, или «0» — в противном случае. При этом пустое множество представляется битовой строкой, целиком состоящей из нулей.

Способ представления массивов без упаковки заключается в том, что каждому элементу отводится одно или несколько целых слов памяти. Другая возможная форма представления — упаковка элементов массива так, чтобы каждый элемент занимал в слове лишь определенное количество битов. Когда массив упакован, задача выбора значения переменной с индексом усложняется.

Реализация работы с файлами предполагает наличие интерфейса компилятора с операционной системой.

Коротко о главном

1. С каждым вызовом процедуры (функции) связывается динамическая область данных фиксированной длины. Когда процедура вызвана, она забирает под свою область данных необходимый объем памяти. Когда выполняется выход из процедуры, память, занятая областью данных, освобождается.
2. Во время выполнения программы возможны обращения к переменным из нескольких областей данных. Поэтому необходимо организовать работу с областями данных так,

чтобы обращения к переменным из разных областей действия (основной программы, процедуры, функции) выполнялись правильно.

3. Для вычисления адреса переменной необходимо знать:
   - уровень вложенности области действия, в которой описана переменная;
   - смещение переменной в области данных.
4. Адрес переменной во время выполнения программы вычисляется по следующей формуле:

   адрес области данных + смещение переменной в этой области данных

5. В машинах с небольшим количеством регистров стек областей данных использует статические и динамические указатели. Чтобы найти адрес переменной во время выполнения программы, необходимо генерировать код для определения адреса области данных, в которой расположена переменная (с помощью статических указателей).
6. Архитектура МКП предоставляет возможность хранить адреса областей данных на *Д*-регистрах, а не в локальной оперативной памяти. В этом случае работа со статическими и динамическими указателями исключается. Чтобы найти адрес переменной во время выполнения программы, достаточно воспользоваться формулой:

   (ДI) + смещение переменной в области данных

   где *I* — уровень вложенности переменной.
7. Элементы массивов обычно размещаются в последовательных ячейках линейной одномерной непрерывной памяти. Значения индексов — это выражения, значения которых определяются во время исполнения программы. Поэтому нужно генерировать код для определения местоположения переменной с индексами во время выполнения программы.
8. Каждый тип данных языка высокого уровня имеет свой собственный способ отображения в форматы представления данных компьютера.

Задания

1. Изобразите стек областей данных для программы **example** в момент вызова процедуры **C**:

```
program example;
  var pr1, pr2:real;
  procedure A;
     var a1, a2, a3: integer;
     procedure B;
        var b1, b2, b3, b4: integer;
        procedure C;
          var c1:real;
          begin ...
              c1:= c1 * b2 * a3 - pr1;
              ...
          end { C };
        begin ... C; ... end { B };
    begin ... B; ... end { A };
  procedure D;
     var d1,d2,d3: integer;
     begin ... A; ... end { D };
  begin ... D; ... end  ... .
```

2. Ответьте на вопрос: как вычисляются адреса переменных в операторе присваивания процедуры **C**?

Глава 10

Генерация кода

10.1. Формирование команд

Как уже говорилось ранее, результат генерации кода — объектная программа — может быть последовательностью машинных команд или программой на языке ассемблера. И поскольку в последнем случае общее время, требуемое для получения программы на машинном языке, увеличивается, в качестве объектного кода мы выберем машинный язык.

Для формирования команд различных форматов мы воспользуемся соответствующими функциями. Будем считать, что имена этих функций отражают смысл команды, а список параметров содержит код операции и, в зависимости от формата команды, номер *С*-, *Р*-, *И*- или *Д*-регистра, смещение или непосредственное данное.

Например, в результате обращения к функции

```
CP_operation (oper, reg1, reg2)
```

будет сформирована команда для *С*- и *Р*-регистров (см. 8.4.1); при этом значение переменной **oper** — это код команды, а значения переменных **reg1** и **reg2** — номера регистров. Уточним описание этой функции:

```
void CP_operation (
   unsigned code                    /* код операции */,
   unsigned regnum1               /* номер С-регистра */,
   unsigned regnum2     /* номер С- или Р-регистра */)
{
   unsigned byte2          /* формируемая команда */ ;
   /* заполнение полей команды */
   byte2 = (code << 9) + (regnum1 << 7) + regnum2;
   /* запись команды в объектный модуль */
   write_2bytecom ( byte2 );
}
```

Теперь рассмотрим формирование команд пересылки данных между регистрами и локальной памятью (см. 8.4.2):

```
/* Пересылка непосредственного данного на регистры */
void mov_RC (
   unsigned numregs                          /* КОЛ */,
   unsigned regtype      /* поле ТО — тип регистра */,
   unsigned regnum         /* поле ОП — номер первого
                                        регистра */,
   long constant       /* непосредственное данное */)
{
    unsigned long hw       /* формируемая команда */;
    /* формирование командного полуслова */
    hw = (0x88 << 24) + (numregs << 15) +
         (0x0A << 10) + (regtype << 8) + regnum ;
    /* запись командного полуслова
                                    в объектный модуль */
    write_halfword ( hw );
    /* запись операндного полуслова
                                    в объектный модуль */
    write_halfword ( constant );
  }

/* Пересылка данных между регистрами
                                  и ЛОП (см. 8.4.2) */
void mov_RM (unsigned numregs                 /* КОЛ */,
   unsigned opcode                  /* код операции */,
   unsigned regtype      /* поле ТО — тип регистра */,
   unsigned regnum                    /* номер первого
                                  регистра группы */,
   unsigned dreg         /* номер дисплей-регистра */,
   unsigned ireg                    /* номер индекса */,
   unsigned long offset                 /* смещение */)
{
  unsigned long
     com_hw                  /* командное полуслово */,
     data_hw               /* операндное полуслово */,
     help1, help2       /* вспомогательные переменные
                        для формирования смещения */;
  /* формирование командного полуслова */
  com_hw = (0x88 << 24) + (numregs << 15) +
       (opcode << 10) + (regtype << 8)+ regnum;
 /* формирование операндного полуслова (см. 8.3) */
   help1 = ( offset/32768ul) & 0xFFul;
   help2 = offset & 0x7FFFul;
```

```
  data_hw = (help1 << 24) + (ireg << 20) +
                        (dreg << 15) + help2;
   /* запись команды в объектный модуль */
   write_halfword ( com_hw );
   write_halfword ( data_hw );
 }
```

Команды передач управления будем генерировать в результате обращения к функциям **branch** и **branch_on_condition**:

```
/* Безусловный переход (см. 8.4.4) */
/* Переход с сохранением адреса возврата */
void branch (
  unsigned code,          /* код операции */
  unsigned inum,          /* номер И-регистра */
  unsigned long offset     /* полусловное
                                  смещение - ПСМ */)
  {
    unsigned long hw;
    /* заполнение командного полуслова: */
    hw = (code << 25) + (inum << 20) + offset;
    /* запись команды в объектный модуль */
    write_halfword (hw);
  }

/* Условный переход по состоянию
                              С-регистра (см.8.4.4) */
void branch_on_condition(
   unsigned cnum              /* номер С-регистра */,
   unsigned condition            /* номер условия */,
   unsigned long offset               /* полусловное
                                   смещение ПСМ */)
  {
     unsigned long hw;
     /* заполнение командного полуслова: */
     hw = (0x54 <<25) + (cnum <<23) +
                 (condition<<20) + offset;
     /* запись команды в объектный модуль */
     write_halfword (hw);
  }
```

Команды остальных форматов генерируются аналогично.

10.2. Промежуточное представление и генерация кода для выражений

Перевод исходной программы в эквивалентную программу на объектном языке мы будем рассматривать как два последовательных преобразования: сначала исходная программа переводится в некоторое промежуточное представление, а затем из него — в последовательность команд на объектном языке. В промежуточном представлении операции располагаются в том порядке, в котором они должны выполняться; это существенно облегчает последующую генерацию кода.

Одним из распространенных способов промежуточного представления исходной программы является *обратная польская запись* (*ОПЗ*). В ней операнды располагаются в том же порядке, что и в исходном выражении, а знаки операций при просмотре слева направо следуют в том порядке, в каком они должны выполняться. Например, выражение: **A+B*C-(Д+E)/Л** в ОПЗ имеет вид: **A B C * + Д E + Л / -**.

В ОПЗ не существует приоритетов операций, здесь также не требуются скобки, так как никогда не возникает сомнений относительно того, какие операнды принадлежат тем или иным знакам операций.

Правило вычисления выражения в ОПЗ состоит в следующем:

```
Установить первый символ выражения текущим;
while (не просмотрены все символы выражения)
 {
   if (текущий символ — знак операции)
     { выполнить операцию над операндами,
       расположенными левее знака операции;
       полученный результат записать на
       место самого левого операнда;
       остальные операнды, участвовавшие в
       операции,и знак операции удалить из ОПЗ;
     }
   установить следующий символ текущим;
 }
```

В итоге запись выражения сократится до одного элемента — результата вычислений.

Основное преимущество ОПЗ перед обычной («инфиксной») записью выражений состоит в том, что описываемые ею действия можно выполнять в процессе однократного безвозвратного просмотра слева направо. Это сближает ОПЗ с записью программы в машинных кодах.

Для перевода выражений из ОПЗ в машинные команды используется *стек операндов* (*СО*). В исходном состоянии СО пуст. Алгоритм перевода состоит в следующем:

```
Установить первый символ выражения текущим;
while (не просмотрена ОПЗ)
  {
   if (текущий символ — операнд)
      занести его в СО;
     else /* текущий символ — операция */
       { сформировать команды, реализующие
         операцию;
         удалить из СО все операнды,
         участвовавшие в операции;
         записать в СО информацию о результате;
       }
   установить следующий символ текущим;
  }
```

Примечание: в однопроходном компиляторе нет необходимости полностью создавать ОПЗ, однако ее построение прослеживается в процессе работы анализатора. Если при анализе программы встречается операнд, то информация о нем заносится в СО. Когда встречается операция, то генерируются соответствующие ей машинные команды; при этом используются верхние элементы стека, которые затем заменяются описанием результата.

В СО хранится информация об операндах следующих классов:

```
#define REFERENCE  1          /*локальная переменная
                                или параметр-значение*/
#define REGISTER   2                    /* регистр */
#define CONSTANT   3    /* непосредственное данное */
#define REF_VAR    4       /* переменная, переданная
                                        по ссылке */
```

Опишем структуру СО:

```
struct stackoperand
 { unsigned loctype              /* класс операнда */;
   TYPEREC *optype                 /* тип операнда */;
   unsigned opsize     /* размер операнда в словах */;
   union
    { /* Регистр */
      struct
       { unsigned whatreg          /* тип регистра */,
                  regnum          /* номер регистра */;
       } opreg;
      /* Непосредственное данное */
      union const_val value             /* значение */;
      /* Локальная переменная или параметр-значение */
      struct memloc
       { unsigned
          dreg                  /* номер Д-регистра */,
          ireg                  /* номер И-регистра */,
          memoff       /* смещение в области данных */;
       } memloc;
      /* Переменная, переданная по ссылке */
      unsigned regnum              /* номер Р-регистра,
                                      содержащего адрес
                                фактического параметра */;
    } location;
  };
```

Введем внешние переменные:

```
struct stackoperand *opstack
```

— адрес начала стека операндов и

```
int up
```

— указатель на вершину стека операндов. Возможный вариант выделения памяти под стек операндов:

```
opstack =
   (struct stackoperand*) malloc
         (sizeof (struct stackoperand)*STACK_SIZE );
```

В исходном состоянии стек операндов пуст. Запись информации об операнде в СО будем выполнять специальными функциями. Опишем две из них:

```
/* запись в СО информации о локальной переменной
   или параметре - значении */
void push_reference (
  TYPEREC *optype          /* указатель на дескриптор
                                 типа операнда */,
  unsigned dreg       /* номер Д-регистра, на котором
                     хранится адрес области данных */,
  unsigned ireg        /* номер И-регистра, на котором
                       хранится динамическое смещение
                                 в области данных */,
  unsigned memoff              /* смещение переменной
                                 в области данных */)
{
   struct stackoperand *ptr;          /* указатель на
                                       вершину СО */
   ptr=(opstack+(++up));
   ptr->loctype=REFERENCE;
   ptr->optype=optype;
   ptr->opsize=optype->size;
   ptr->location.memloc.dreg=dreg;
   ptr->location.memloc.ireg=ireg;
   ptr->location.memloc.memoff=memoff;
}

/* запись в СО информации об операнде,
    значение которого хранится на регистре */
 void push_register (
  TYPEREC *optype          /* указатель на дескриптор
                                 типа операнда */,
  unsigned  whatreg              /* тип регистра */,
  unsigned  regnum            /*  номер регистра */)
{ struct stackoperand *ptr              /* указатель
                                 на вершину СО */;
   ptr = (opstack + (++up));
   ptr -> loctype = REGISTER;
   ptr -> optype = optype;
   ptr -> opsize = optype -> size;
   ptr -> location.opreg.whatreg = whatreg;
   ptr -> location.opreg.regnum = regnum;
}
```

Теперь добавим в функции анализа конструкций языка действия по формированию СО и генерации кода. В качестве примера рассмотрим описание функций компиляции конструкций <*слагаемое*> и <*множитель*> (в этом и последующих примерах для краткости изложения опустим операторы, выполняющие нейтрализацию синтаксических ошибок).

Примечание: если в процессе анализа программы компилятор обнаружит ошибку, то нет смысла выполнять дальнейшую генерацию кода. Поэтому введем внешнюю переменную

```
unsigned nocode
```

значение которой равно **false** при отсутствии ошибок и **true** — в противном случае. В дальнейшем все действия по генерации кода мы будем сопровождать проверкой:

```
  if (!nocode)
    { /* действия по генерации кода */
        ...
    }
```

```
TYPEREC *term (unsigned *followers)
/* компиляция слагаемого */
{
 TYPEREC *ex1type,
         *ex2type         /*указатели на дескрипторы
                                типов множителей */;
 unsigned operation              /* код операции */;
 unsigned ptra [SET_SIZE]     /* внешние символы */;
 set_disjunct (op_mult, followers, ptra);
 ex1type = factor (ptra);
 while (belong (symbol, op_mult))
  { operation = symbol;           /* запоминаем знак
                                        операции */
    nextsym( );  ex2type = factor (ptra);
    ex1type = term_test (ex1type,
                         ex2type, operation);
      /* генерация кода для операции
                                    типа умножения */
    if (!nocode)
        multop (operation, ex1type);
  }
 return (ex1type);
}
```

Запись информации об операндах выражения в СО выполняется в функции **factor**:

```
TYPEREC  *factor (unsigned *followers)
/* компиляция конструкции  <множитель>*/
  {
   union const_val value     /* значение константы */;
   TYPEREC *exptype         /* указатель на дескриптор
                                  типа множителя */;
   NODE *node;             /* указатель на вершину ТИ,
          соответствующую текущему идентификатору */
   unsigned
    after_express[SET_SIZE];     /* внешние символы */
   switch (symbol)
    { case leftpar:
                ...
       break;
      case intc:                  /* целая константа */
       exptype = inttype;
       if (!nocode)
         { value.intval = nmb_int;
            /* запись в СО информации
                                  о целой константе */
           push_constant (inttype, value);
         }
         nextsym ( );
         break;
       case charc:           /* символьная константа */
         exptype = chartype;
         if (!nocode)
           { value.intval = onesymbol;
             /* запись в СО информации о
             символьной константе */
             push_constant (chartype, value);
           }
         nextsym();
         break;
       case ident:               /* имя функции или имя
                           переменной, или имя поля */
           ...
              case VARS:           /*  имя переменной */
                 exptype=variable( followers );
                 break;
           ...
   }
  return ( exptype );
}
```

При обработке прикладного вхождения переменной информация о ее местоположении во время выполнения заносится в стек операндов в функции **variable**, анализирующей конструкцию *<переменная>*. Для простой локальной переменной это:

- номер *Д*-регистра, который равен уровню вложенности переменной;
- номер *И*-регистра, заданный числом 17, что означает отсутствие индексации;
- значение смещения, хранящееся в поле **offset** структуры **treenode**.

Эта информация записывается в СО следующим образом:

```
/* переменная node содержит указатель
                                    на вершину ТИ */
if (!nocode)
  push_reference
           (node->idtype,
            node->casenode.vars.staticlevel, 017,
            node->casenode.vars.offset);
```

Для дальнейшего изложения воспользуемся функцией

```
short type_analyze( )
```

которая анализирует типы операндов в вершине СО и возвращает одно из следующих значений:

INT_INT, INT_REAL, REAL_INT, REAL_REAL, SET_SET

в соответствии с типами предпоследнего и последнего элементов СО, а также функцией:

```
void op_analyze
     (unsigned kindreg          /* тип регистра -
                              входной параметр */,
      &numreg                   /* номер регистра -
                             выходной параметр */),
```

которая:

- анализирует местоположение операнда в вершине СО и генерирует команды его пересылки в регистр с типом **kindreg** и номером **numreg**;
- реализует распределение регистров;
- удаляет элемент из СО.

Наконец, после введения вспомогательных функций опишем функцию **multop**, генерирующую код для операций типа умножения. К моменту формирования команды в СО содержится информация о местоположении операндов во время выполнения программы, поэтому для получения окончательного машинного кода нам остается заменить знак операции машинными командами:

```
multop (
  unsigned operation       /* код операции */,
  TYPEREC *exptype          /* указатель на дескриптор
                           типа результата операции */)
  { unsigned  reg_c1           /* номер С-регистра */,
              reg_c2           /* номер С-регистра */,
              reg_p            /* номер Р-регистра */;
    short optypes                /* информация о типах
                               операндов в вершине  СО */;
    optypes = type_analyze ();
    op_analyze (C_REG, reg_c1);
    op_analyze (P_REG, reg_p);
    switch (operation)
      { case andsy:
          CP_operation (047, reg_c1, reg_p);
          break;
        case star:
          switch (optypes)
            { case INT_INT:
                CP_operation ( 032, reg_c1, reg_p);
                break;
              case INT_REAL:
                reg_c2 = get_C_reg ( );        /* запрос
                                              С-регистра */
                CP_operation (013, reg_c2, reg_p);
                CP_operation (005, reg_c1, reg_c2);
                free_C_reg (reg_c2);     /* освобождение
                                              С-регистра */
                break;
              case REAL_INT:
                CP_operation (013, reg_c1, reg_c1);
              case REAL_REAL:
                CP_operation (005, reg_c1, reg_p);
                break;
```

```
            case SET_SET:  ...
          }
        break;
       case slash:  ...
       case modsy:  ...
       case divsy:  ...
     }
     push_register (exptype, C_REG, reg_c1);
     free_P_reg (reg_p);              /* освобождение
                                        P-регистра */
    }
}
```

Функции, реализующие генерацию кода для других операций языка Паскаль, описываются аналогично.

10.3. Промежуточное представление и генерация кода для операторов

В предыдущем разделе мы познакомились с основными свойствами ОПЗ для выражений. Именно на этих свойствах основано использование ОПЗ при трансляции операторов. Для перевода в ОПЗ ряда операторов необходимо ввести операции, которых в исходной записи нет. Набор дополнительных операций выбирается исключительно из соображений простоты перевода в ОПЗ, а затем — в машинный код.

ОПЗ для условного оператора

if A **then** S1 **else** S2

имеет вид:

A Label_1 bf S1 Label_2 branch Label_1: S2 Label_2:

Здесь операция **bf** имеет два операнда: первый — логическое выражение, второй — метка. Если логическое выражение истинно, то операция **bf** пропускается, а если ложно, то происходит переход на метку. У операции **branch** имеется лишь один операнд — метка. Результат операции **branch** — переход на эту метку.

Неполный условный оператор

if A **then** S1

в ОПЗ записывается так:

A Label_1 bf S1 Label_1:

К моменту генерации команд перехода значение поля *ПСМ* (см. п. 8.4.4) неизвестно, так как еще не определен адрес перехода. Чтобы в дальнейшем доформировать эти команды, введем две переменные: **unsigned offset1** и **offset2** — полусловное смещение команд условного и безусловного перехода соответственно. Тогда алгоритм генерации кода для условного оператора будет иметь вид:

1) сгенерировать команды пересылки значения выражения *A* на *C*-регистр;
2) запомнить текущее значение *ПСМ* в переменной **offset1**;
3) сгенерировать команду условного перехода по значению *C*-регистра на метку **Label_1**;
4) сгенерировать код для оператора **S1**;
5) запомнить текущее *ПСМ* в переменной **offset2**;
6) сгенерировать команду безусловного перехода на оператор, следующий сразу после **if**;
7) доформировать команду условного перехода;
8) сгенерировать код для оператора **S2**;
9) доформировать команду безусловного перехода.

Таким образом, функцию, реализующую синтаксический, семантический анализ, а также генерацию кода для *условного оператора*, можно записать так:

```
void ifstatement (unsigned *followers)
/* компиляция условного оператора */
{ TYPEREC *exptype          /* указатель на дескриптор
                                  типа выражения */;
  unsigned ptra [SET_SIZE]       /* множество внешних
                                        символов */,
  offset1, offset2             /* полусловное смещение
                                     команд перехода */,
  reg_c                        /* номер C-регистра */;
  nextsym ( );
    /* формирование множества внешних символов
     для конструкции <выражение> */
  set_disjunct(af_iftrue, followers, ptra);
  exptype = expression (ptra);
  if (!compatible (exptype, booltype))
    error (135);
```

```
  /*генерация команд пересылки значения
    выражения на С-регистр*/
  if (!nocode)
    {op_analyze (C_REG, reg_c);
      /* запомнить текущее значение ПСМ */
      offset1 = comoff;          /* внешняя переменная
                    comoff содержит смещение текущей
                 команды в строке программного кода */;
      /* генерация команды условного перехода */
      branch_on_condition
                  (reg_c   /* номер С-регистра */,
                   0            /* код условия */,
                   0  /* полусловное смещение */ );
      }
  accept( thensy );

  /* формирование множества внешних символов
    для конструкции <оператор>*/
  set_disjunct (af_iffalse, followers, ptra);
  statement(ptra);
  if (symbol == elsesy)
     {if (!nocode)
        { offset2 = comoff;
          /* генерация команды безусловного
                                      перехода */
          branch ( 0123 /* код операции */,
                    017 /* номер И-регистра */,
                    0 /* полусловное смещение */);
          /* доформирование команды, расположенной
             по смещению offset1 — запись в поле ПСМ
             значения comoff */
          psm_for_branch ( offset1 );
         }
      nextsym ( );
      statement ( followers );
      if (!nocode)
          psm_for_branch ( offset2 );
     }
     else
      if (!nocode)
        psm_for_branch ( offset1 );
}
```

ОПЗ для *оператора цикла с предусловием*

while A **do** S

имеет вид:

```
Label_2: A Label_1 bf S Label_2 branch Label_1
```

```
void whilestatement (unsigned *followers)
/* компиляция оператора цикла с предусловием */
{ TYPEREC *exptype          /* указатель на дескриптор
                                 типа выражения */;
  unsigned ptra [SET_SIZE]       /* множество внешних
                                       символов */,
  offset1, offset2             /* полусловное смещение
                                    команд перехода */,
  reg_c                        /* номер С-регистра */;
  nextsym ( );
    /* формирование множества внешних символов
         для конструкции <выражение> */
  set_disjunct ( af_while, followers, ptra );
  if (! nocode) offset1 = commof;
  exptype = expression ( ptra );
  if (! compatible(exptype, booltype))
     error (135);
  /*генерация команд пересылки значения
                               выражения на С-регистр*/
  if (!nocode)
    { op_analyze(C_REG, reg_c);
     /* запомнить текущее значение ПСМ */
     offset2 = comoff;           /* внешняя переменная
                     comoff содержит смещение текущей
                команды в строке программного кода */;
     /* генерация команды условного перехода */
     branch_on_condition
                 (reg_c  /* номер С-регистра */,
                  0      /* код условия */,
                  0      /* полусловное смещение */ );
    }
  accept (dosy);
  statement (followers);
  if (!nocode)
```

```
      {
        /* генерация команды безусловного перехода */
        branch
          (0123                    /* код операции */,
           017                  /* номер И-регистра */,
           offset1        /* полусловное смещение */);
      /* доформирование команды, расположенной
         по смещению offset2 — запись в поле
         ПСМ значения comoff */
      psm_for_branch (offset1);
      }
}
```

ОПЗ *оператора присваивания*

```
V := Expr
```

имеет вид:

```
V Expr :=
```

Прежде чем описывать действия по генерации кода для оператора присваивания, уточним функцию **statement**, соответствующую конструкции ***<оператор>*** (см. также п. 6.3):

```
void statement(unsigned *followers)
/*компиляция конструкции <оператор> */
{
 NODE *firstid,   /*указатель на вершину дерева*/
 if(symbol==intc)
   { /* обработка метки */
       ...
     nextsym(); accept(colon);
   }
 switch(symbol)
   { case ident:
      /* находимся в контексте оператора with
         и идентификатор — имя поля */
     if (localwith!=NULL &&
           searchfield()!=NULL)
       /* обработка оператора присваивания, в
          левой части которого — имя поля */
       assignment(followers, NULL);
```

```
  else
    {
     /* возможно, имя переменной или
                      функции, или процедуры */
     /* ищем имя в ТИ */
     firstid = searchident
         (hashresult,addrname, set_VARFUNPR);
     if (firstid != NULL)      /* нашли имя в ТИ */
       switch (firstid->class)        /* проверяем
                      способ использования имени */
         { case VARS:
           case FUNCS:
           /* оператор присваивания,в левой
              части которого — имя переменной
              или имя функции */
              assignment (followers, firstid);
              break;
            case PROCS: /* вызов процедуры */
              CallProc (followers,firstid);
              break;
         }
         else          /* имя стандартной процедуры
                          или ошибка*/
    StandardProc(followers);
    }
 break;
case beginsy:
    compoundstatement(followers);break;
case
    ifsy:ifstatement(followers);break;
case
    whilesy:whilestatement(followers);break;
case
    repeatsy:repeatstatement(followers);break;
case
    forsy:forstatement(followers);break;
case
    casesy:casestatement(followers);break;
case
    withsy:withstatement(followers);break;
case
    gotosy:gotostatement(followers);break;
```

```
    case semicolon:
    case endsy:
    case untilsy:
    case elsesy:break;/*в случае пустого оператора*/
  }
}
```

При генерации кода для оператора присваивания важно учитывать различные варианты местоположения выражения во время выполнения программы (память, регистр, непосредственное данное). Кроме того, нужно помнить, что в левой части может находиться как переменная, так и имя функции. Чтобы продемонстрировать основные принципы генерации кода для присваивания, рассмотрим случай, когда в левой части оператора находится переменная, а выражение располагается в регистре:

```
void assignment(
   unsigned *followers,
   NODE *first /*указатель на вершину дерева */)
/*компиляция оператора присваивания */
{
 unsigned  ptra[SET_SIZE];       /* внешние символы */
 TYPEREC *exptype;          /* указатель на дескриптор
                                    типа выражения */
 TYPEREC *vartype;          /* указатель на дескриптор
                                    типа переменной*/
 struct stackoperand
            *expptr, *varptr;        /* указатели на
                                     элементы СО */
/*формирование внешних символов
  для конструкции <переменная> */
 set_disjunct(af_assignment,followers,ptra);
/*рассмотрен случай, когда в левой части оператора —
   имя поля или имя переменной, а именно first
   равно NULL или first->class равно VARS */
 vartype = variable(ptra);              /* обработка
                           конструкции <переменная> */
 accept(assign);
 exptype=expression(followers);
 /* проверка совместимости типов переменной и
                                          выражения */
 if  (!compatible_for_assign (vartype, exptype))
    Error(145);
```

```
if (!nocode)
 { /* получить из СО информацию о выражении */
    expptr = opstack+up;
   /* получить из СО информацию о переменной */
   varptr = opstack+up-1;
   switch(varptr->loctype)
     {case REFERENCE:          /* локальная переменная
                                  или параметр-значение */
        /* генерация команд пересылки значения
           выражения в память */
        switch (expptr->loctype)
          {case REGISTER:    /* выражение —
                                    на регистре */
            if
             /*присваивание вещественной
               переменной целого или
               ограниченного на целом значения */
               (vartype==realtype &&
               (exptype==inttype ||
                exptype->typecode==LIMITEDS))
              {
              /*генерация  команд преобразования
                целого значения в вещественное
                и загрузки его на регистр */
                CP_operation(013,1,expptr->
                             location.opreg.regnum);
                CP_operation(036,1,expptr->
                             location.opreg.regnum);
               }
             /* генерация команды пересылки из
                регистра в память */
              movRM
               (expptr->opsize-1, 030, P_REG,
                expptr->location.opreg.regnum,
                varptr-> location.memloc.dreg,
                varptr-> location.memloc.ireg,
              varptr-> location.memloc.memoff);
              break;
          case CONSTANT: ...  break;
          case REFERENCE: ...  break;
          case REF_VAR: ... break;
         } /* switch for expression */
```

```
        break;
      case  REF_VAR:        /* в левой части —
           параметр,переданный по ссылке */
                          ...
        break;
     } /* switch for variable */
up-=2; /* информация о переменной и выражении
          удаляется из СО */
 }
}
```

ОПЗ для *вызова процедуры*

```
P(X1,  X2,...,XN)
```

может быть представлена так:

update X1 write ... XN write adr go_back

Здесь операция **update** — это обновление информации, связанной со стеком областей данных. В момент активации процедуры N-го уровня вложенности необходимо сгенерировать команды, обновляющие информацию, связанную со стеком областей данных. Эти команды реализуют выполнение следующих действий:

1) сохранение всех *И*-регистров и регистра *ДN* в аппаратурный стек;
2) коррекцию адреса начала: в регистр *ДN* надо поместить дескриптор, взятый из *Д*33, у которого поле начала увеличено на значение *И*16 (в регистре *И*16 хранится адрес первой свободной ячейки программного стека);
3) коррекцию длины сегмента: в регистре *ДN* надо установить значение поля **<длина>** равным размеру области данных процедуры уровня N;
4) увеличение значения *И*16 на размер области данных процедуры;
5) преобразование виртуального дескриптора, содержащегося в регистре *ДN*, в физический.

В результате выполнения операции **write** значение или адрес фактического параметра (в зависимости от способа передачи) записывается в область данных процедуры по смещению соответствующего формального параметра. Информация о способе передачи должна храниться в ТИ.

Для передачи управления процедуре (операция **go_back**) необходимо знать адрес начала ее машинного кода (**adr**). Поэтому перед трансляцией раздела операторов процедуры текущее полусловное смещение запоминается в вершине ТИ, соответствующей этой процедуре. В связи с этим еще раз уточним структуру **treenode**:

```
struct treenode
 {
  unsigned hashvalue    /* значение hash – функции */;
  char *idname        /* адрес имени в таблице имен */;
  unsigned class           /* способ использования */;
  TYPEREC  *idtype       /* указатель на
                                  дескриптор типа */;
  union
   { /* для констант */
     union const_val constvalue;        /* значение */
     /* для процедур ( функций ) */
     struct
        { struct idparam *param;          /* указатель
                        на информацию о параметрах */
          int forw;                     /* информация об
                              опережающем описании */;
          int count_locals            /* размер области
                        данных процедуры (функции) */;
          int begin_code      /* указатель на
                                     начало кода */;
        } proc;
       /* для переменных */
       struct
         { unsigned staticlevel;          /* уровень
                           вложенности переменной */,
           offset;              /* смещение переменной
                                 в области данных */;
           unsigned param_var          /* =TRUE,  если
                      переменная является параметром,
                              переданным по ссылке */;
        } vars;
    } casenode;
  struct treenode *leftlink ;
  struct treenode *rightlink;
 };
```

Рассмотрим последовательность генерируемых команд для программы:

```
program example;
   var g,f: real;
   procedure A ( x: real; var y: real );
     begin          ...          end;
 begin A ( g, f );   ...      end.
```

Код процедуры A:

- команда **Начало блока;**
- команда **Сдвиг зоны обработки с одновременным изменением ее длины;**
- команда пересылки дескриптора из *Д*31 в *Д*2 (*Д*31 — вспомогательный регистр, содержащий адрес области данных процедуры **A;** загрузка регистра *Д*31 выполняется в основной программе в момент вызова процедуры **A);**
- команды, реализующие операторы процедуры **A;**
- команда **Возврат.**

Код основной программы:

- команда **Начало блока;**
- команда **Сдвиг зоны обработки с одновременным изменением ее длины;**
- команда пересылки дескриптора из *Д*33 в *Д*1;
- команда **Коррекция длины сегмента:** в *Д*1 загружается дескриптор из *Д*1, у которого поле длины равно размеру области данных основной программы;
- команды записи виртуального дескриптора из *Д*1 по нулевому смещению области данных основной программы;
- команда преобразования виртуального дескриптора из *Д*1 в физический;
- команда загрузки в *И*16 адреса первой свободной ячейки в стеке областей данных;
- код для вызова процедуры **A:**
  - — команда **Коррекция адреса начала:** в *Д*31 загружается дескриптор, взятый из *Д*33, у которого поле начала увеличено на значение из *И*16;
  - — команда **Коррекция длины сегмента:** в *Д*31 загружается дескриптор, взятый из *Д*31, у которого поле длины устанавливается равным размеру области данных процедуры **A;**

— команды записи виртуального дескриптора из *Д*31 по нулевому смещению области данных процедуры **A**;
— команда преобразования виртуального дескриптора из *Д*31 в физический;
— команда загрузки на *И*16 адреса первой свободной ячейки в стеке областей данных;
— команды загрузки значений или адресов фактических параметров в область данных, адрес которой находится в *Д*31;
— команда **Переход с возвратом** (передача управления на начало кода процедуры **A**);
— команда **Возврат**.

Теперь опишем функцию компиляции вызова процедуры.

```
void call_proc (
   unsigned *followers,
   struct treenode *ptr_proc;          /* указатель на
                    вершину ТИ вызываемой процедуры*/)
{
  unsigned reg_P;                  /* номер Р-регистра */
  unsigned ptra [SET_SIZE];        /* внешние символы */
  struct idparam *ptrparam;              /* указатель на
                          информацию о параметре в ТИ */
  TYPEREC *typevar,  /* тип фактического параметра */
  *h_ptr;               /* тип формального параметра */
  nextsym( );
  if (!nocode)
   { /* обновление информации, связанной
          со стеком областей данных */
     /* коррекция адреса начала;
          см. 8.4.6.1 и 10.1 */
     mov_RM (0, 06, 10, 031, 033, 016, long(0));
     /* коррекция длины сегмента */
     mov_RM (0, 05, 10, 031, 031, 017,
        ptr_proc -> casenode.proc.count_locals);
       /* запись виртуального дескриптора из Д31
          по нулевому смещению области данных
          процедуры и преобразование виртуального
          дескриптора в физический */
     reg_P = get_P_reg ( )     /* запрос Р-регистра */;
     mov_PD (031 /* оп/рез */,
           1 /* КОП */, 0 /* ТО */,
```

```
        reg_P /* ОП * /);     /*  см. 8.4.6.2  */
   mov_PD (031, 2, 2, 031);
   mov_RM (0, 030, 0, reg_P, 031, 017, 0);
   /* загрузка на Р-регистр размера области данных */
   mov_RC
     (0, 0, reg_P, ptr_proc ->
      casenode.proc.count_locals);  /* см. 8.4.2 */
   /* загрузка на И16 адреса первой свободной
   ячейки в стеке областей данных */
   index_operation
      (016   /* Иоп */ , 016 /* Ирез */,
       0 /* АО */, 00 /* ТО */,
       reg_P /* ОП */ );  /* см. 8.4.3.1 */
   free_P_reg (reg_P);
}
  /* анализ параметров и генерация кода
     для загрузки  информации о параметрах
     в область данных процедуры */
set_disjunct (af_fact_param, followers, ptra);
if (symbol == leftpar)          /* есть фактические
                                        параметры */
  {
   ptrparam  =
   ptrproc -> casenode.proc.param       /* указатель
                              на информацию о первом
                         формальном параметре в ТИ */
   if (ptrparam == NULL)       /* формальных
                                    параметров нет */
      error (126);
     else
       { do
          { nextsym ( );
            /* получаем указатель на дескриптор
               типа формального параметра */
            h_ptr = ptrparam -> typeparam;
            switch (ptrparam -> mettransf)  /* способ
                                передачи параметра */
               {
                 case paramvar:
                    /* получаем указатель на
                       дескриптор типа
                       формального параметра */
```

```
                    typevar=variable (ptra);
                    if (!compatible (h_ptr,
                                     typevar ))
                        error(189);
                    /* генерация кода для загрузки
                      адреса фактического параметра */
                    gen_param_var(ptrparam);
                    break;
                  case paramval:
                        ...
                 }
               /* переходим к анализу
                   следующего параметра */
                ptrparam = ptrparam -> linkparam;
               }
              while (symbol ==comma
                          && ptrparam != NULL);
              if (ptrparam != NULL)
                error (126);  /* фактических параметров
                                 меньше, чем формальных */
              accept (rightpar);
        }
   } /* if (symbol == leftpar) */
    else
      {   ...    }
  if (!nocode)
  /* генерация команды «Переход с сохранением
         адреса возврата»; см. 8.4.4 */
    branch(0115 /* КОП */, 017 /* И */,
    ptrproc->casenode.proc.begin_code /* ПСМ */ );
}
```

Для загрузки информации о фактических параметрах необходимо знать смещение соответствующего формального параметра относительно начала области данных. Поэтому дополним структуру **idparam** еще одним полем. Тогда окончательный вид этой структуры будет таким:

```
struct idparam
  {
   TYPEREC *typeparam;    /* адрес дескриптора типа */
   int mettransf;                /* способ передачи */
```

```
    struct prfun_as_par  *par;          /* указатель на
                                информацию о параметрах
                        параметра-процедуры (функции) */
    unsigned par_offset;          /* смещение параметра
                    относительно начала области данных */
    struct idparam *linkparam ;
  };

void gen_param_var
   (struct idparam  *par_ptr           /* указатель на
                               информацию о параметре */)
   /* генерация кода для загрузки адреса фактического
  параметра  в область данных вызываемой процедуры */
  {
    unsigned paramoffset;         /* смещение параметра
                    относительно начала области данных */
    struct memloc h_loc;         /* информация об адресе
                                 локальной переменной -
                        номер Д-, И-регистра и смещение */
    struct stackoperand  *varptr;       /* указатель на
                               вершину стека операндов */
    if (!nocode)
     {
       paramoffset = par_ptr -> paroffset;
       /* читаем и удаляем с вершины СО информацию
          о фактическом параметре, передаваемом
          переменной */
       varptr = opstack + up;  up--;
       if (varptr -> loctype == REF_VAR)
         {
           mov_RM(0,030,0,varptr->location.regnum,
                             031, 017, paramoffset);
           free_P_reg(varptr -> location.regnum);
         }
          else
             {
               /* varptr -> loctype = REFERENCE */
               h_loc = varptr -> location.memloc;
               /* виртуальный дескриптор области данных
                  фактического параметра загружаем во
                  вспомогательный регистр Д30 */
               mov_RM (0, 010, 2, 030,
                             h_loc.dreg, 017, 0);
```

```
            /* коррекция адреса начала
               дескриптора из Д30 */
            mov_RM (0, 06, 2, 030, 030,
                              h_loc.ireg, h_loc.memoff);
            /*  запись содержимого Д30 по смещению
               формального параметра */
            mov_RM (0, 030, 2, 030, 031,
                              017, paramoffset);
          }
  }
}
```

Коротко о главном

1. Для формирования команд различных форматов используются соответствующие функции. Имена этих функций отражают смысл команды, а список параметров содержит код операции и, в зависимости от формата команды, номера регистров, смещения или непосредственные данные.
2. Перевод исходной программы в эквивалентную программу на объектном языке включает в себя два последовательных преобразования. Сначала исходная программа переводится в промежуточное представление, а затем оно преобразуется в последовательность команд на объектном языке.
3. В промежуточном представлении операции располагаются в том порядке, в котором они должны выполняться. Это существенно облегчает последующую генерацию кода.
4. Одним из распространенных способов промежуточного представления исходной программы является обратная польская запись (ОПЗ). В ОПЗ операнды располагаются в том порядке, что и в исходном выражении, а знаки операций при просмотре слева направо следуют в том порядке, в каком они должны выполняться.
5. В однопроходном компиляторе нет необходимости полностью создавать ОПЗ, однако ее построение прослеживается в процессе работы анализатора.
6. Для перевода выражений из ОПЗ в машинные команды используется стек операндов (СО).
7. В СО хранится информация об операндах следующих классов: локальная переменная или параметр-значение, регистр, непосредственное данное, переменная, переданная по ссылке.

8. Запись информации об операндах выражения в СО выполняется в функции анализа конструкции <*множитель*>.
9. Для перевода в ОПЗ ряда операторов необходимо ввести операции, которых в исходной записи нет. Набор дополнительных операций выбирается исключительно из соображений простоты перевода в ОПЗ и затем — в машинный код.

Задание

1. Дополните анализатор действиями по генерации кода.
2. Разработайте набор тестов для тестирования генератора кода.

Литература

1. *Йенсен К., Вирт Н.* Паскаль. Руководство для пользователя. — М.: Финансы и статистика, 1989.
2. *Керниган Б., Ритчи Д. А.* Язык программирования Си. — М.: Издательский дом «Вильямс», 2006.
3. *Ахо А., Сети Р., Ульман Дж.* Компиляторы. Принципы, технологии, инструменты. — М.: Издательский дом «Вильямс», 2003.
4. *Хантер Р.* Основные концепции компиляторов. — М.: Издательский дом «Вильямс», 2002.
5. *Welsh J., Mcklag M.* Structered System Programming. — London: Prentice-Hall International, 1980.
6. *Бяков А. Ю., Кропачев Ю. А.* Модульный конвейерный процессор. — М.: ИТМ и ВТ РАН, 1990.
7. *Майерс Г.* Архитектура современных ЭВМ. В 2 кн. — М.: Мир, 1985.

Приложение 1

Синтаксис стандарта языка Паскаль

Формы Бэкуса—Наура (БНФ)

<программа>::=**program** <имя>(<имя файла>{,<имя файла>});
<блок>.

<имя файла>::=<имя>

<имя>::=<буква>{<буква>|<цифра>}

<блок>::=<раздел меток><раздел констант><раздел типов>
<раздел переменных><раздел процедур и функций>
<раздел операторов>

<раздел меток>::=<пусто>|**label** <метка>{,<метка>};

<метка>::=<целое без знака>

<раздел констант>::=<пусто>|**const** <определение константы>;
{<определение константы>;}

<определение константы>::=<имя>=<константа>

<константа>::=<число без знака>|<знак><число без знака>|
<имя константы>|<знак><имя константы>|<строка>

<число без знака>::=<целое без знака>|
<вещественное без знака>

<целое без знака>::=<цифра>{<цифра>}

<вещественное без знака>::=<целое без знака>.<цифра>{<цифра>}|
<целое без знака>.<цифра>{<цифра>}E<порядок>|
<целое без знака>E<порядок>

<порядок>::=<целое без знака>|<знак><целое без знака>

<знак>::=+|-

<имя константы>::=<имя>

<строка>::='<символ>{<символ>}'

<раздел типов>::=<пусто>|**type** <определение типа>;
{<определение типа>;}

<определение типа>::=<имя>=<тип>

<тип>::=<простой тип>|<составной тип>|<ссылочный тип>

```
<простой тип>::=<перечислимый тип>|<ограниченный тип>|
    <имя типа>

<перечислимый тип>::=(<имя>{,<имя>})

<ограниченный тип>::=<константа>..<константа>

<имя типа>::=<имя>

<составной тип>::=<неупакованный составной тип>|
   packed <неупакованный составной тип>

<неупакованный составной тип>::=<регулярный тип>|
   <комбинированный тип>|<множественный тип>|
   <файловый тип>

<регулярный тип>::= array[<простой тип>{,<простой тип>}]
   of <тип компоненты >

<тип компоненты>::=<тип>

<комбинированный тип>::=record <список полей> end

<список полей>::=<фиксированная часть>|
   <фиксированная часть><вариантная часть>|
   <вариантная часть>

<фиксированная часть>::=<секция записи>{;<секция записи>}

<секция записи>::=<имя поля>{,<имя поля>}:<тип>|<пусто>

<вариантная часть>::= case <поле признака><имя типа> of
    <вариант>{;<вариант>}

<поле признака>::=<имя поля>:|<пусто>

<вариант>::=<список меток варианта>:(<список полей>)|
    <пусто>

<список меток варианта>::=<метка варианта>
   {,<метка варианта>}

<метка варианта>::=<константа>

<множественный тип>::= set of <базовый тип>

<базовый тип>::=<простой тип>

<файловый тип>::= file of <тип>

<ссылочный тип>::=↑<имя типа>

<раздел переменных>::= var <описание однотипных переменных>;
   {<описание однотипных переменных>;}|<пусто>

<описание однотипных переменных>::=<имя>{,<имя>}:<тип>

<раздел процедур и функций>::=
   {<описание процедуры или функции>;}
```

<описание процедуры или функции>::=<описание процедуры> |
<описание функции>

<описание процедуры>::=<заголовок процедуры><блок>

<заголовок процедуры>::= **procedure** <имя>;| **procedure** <имя>
(<раздел формальных параметров>{;<раздел формальных
параметров>});

<раздел формальных параметров>::=<группа параметров>|
var <группа параметров>| **function** <группа параметров>|
procedure <имя>{,<имя>}

<группа параметров>::=<имя>{,<имя>}:<имя типа>

<описание функции>::=<заголовок функции><блок>

<заголовок функции>::= **function** <имя>:<тип результата>;|
function <имя>(<раздел формальных параметров>
{;<раздел формальных параметров>}):<тип результата>;

<тип результата>::=<имя типа>

<раздел операторов>::=<составной оператор>

<оператор>::=<непомеченный оператор>|<метка>
<непомеченный оператор>

<непомеченный оператор>::=<простой оператор>|
<сложный оператор>

<простой оператор>::=<оператор присваивания>|
<оператор процедуры>|<оператор перехода>|
<пустой оператор>

<оператор присваивания>::=<переменная>:=<выражение>|
<имя функции>:=<выражение>

<переменная>::=<полная переменная>|
<компонента переменной>|<указанная переменная>

<полная переменная>::=<имя переменной>

<имя переменной>::=<имя>

<компонента переменной>::=<индексированная переменная>
|<обозначение поля>|<буфер файла>

<индексированная переменная>::=<переменная-массив>
[<выражение>{,<выражение>}]

<переменная-массив>::=<переменная>

<обозначение поля>::=<переменная-запись>.<имя поля>

<переменная-запись>::=<переменная>

<имя поля>::=<имя>

<буфер файла>::=<переменная-файл>↑

<переменная-файл>::=<переменная>

<указанная переменная>::=<переменная-ссылка>↑

<переменная-ссылка>::=<переменная>

<выражение>::=<простое выражение>|<простое выражение>
<операция отношения><простое выражение>

<операция отношения>::==|<>|<|<=|>=|>|**in**

<простое выражение>::=<знак><слагаемое>
{<аддитивная операция><слагаемое>}

<аддитивная операция>::=+|-|**or**

<слагаемое>::=<множитель>{<мультипликативная операция>
<множитель>}

<мультипликативная операция>::=*|/|**div**|**mod**|**and**

<множитель>::=<переменная>|<константа без знака>|
(<выражение>)|<обозначение функции>|<множество>|
not <множитель>

<константа без знака>::=<число без знака>|<строка>|
<имя константы>|**nil**

<обозначение функции>::=<имя функции>|<имя функции>
(<фактический параметр>{,<фактический параметр>})

<имя функции>::=<имя>

<множество>::=[<список элементов>]

<список элементов>::=<элемент>{,<элемент>}|<пусто>

<элемент>::=<выражение>|<выражение>..<выражение>

<оператор процедуры>::=<имя процедуры>|<имя процедуры>
(<фактический параметр>{,<фактический параметр>})

<имя процедуры>::=<имя>

<фактический параметр>::=<выражение>|<переменная>|
<имя процедуры>|<имя функции>

<оператор перехода>::= **goto** <метка>

<пустой оператор>::=<пусто>

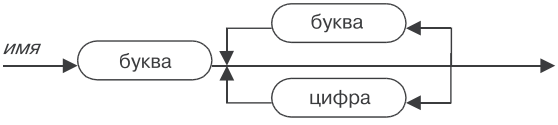
<пусто>::=

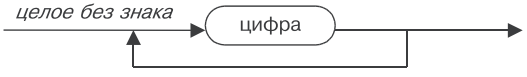
<сложный оператор>::=<составной оператор>|
<выбирающий оператор>|<оператор цикла>|
<оператор присоединения>

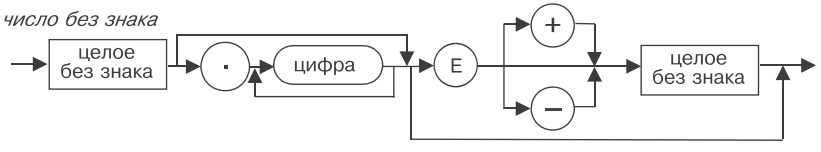
<составной оператор>::= **begin** <оператор>{;<оператор>} **end**

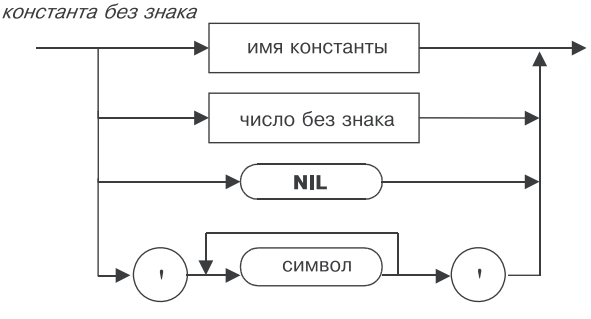
```
<выбирающий оператор>::=<условный оператор>|
   <оператор варианта>

<условный оператор>::= if <выражение> then <оператор>|
   if <выражение> then <оператор> else <оператор>

<оператор варианта>::= case <выражение> of
   <элемент списка вариантов>
   {;<элемент списка вариантов>} end

<элемент списка вариантов>::=<список меток варианта>:
    <оператор>|<пусто>

<список меток варианта>::=<метка варианта>
   {,<метка варианта>}

<оператор цикла>::=<цикл с предусловием>|
   <цикл с постусловием>|<цикл с параметром>

<цикл с предусловием>::= while <выражение> do <оператор>

<цикл с постусловием>::= repeat <оператор>{;<оператор>}
    until <выражение>

<цикл с параметром>::= for <параметр цикла>:=<выражение>
   <направление><выражение> do <оператор>

<параметр цикла>:=<имя>

<направление>::= to|downto

<оператор присоединения>::= with
   <список переменных-записей> do <оператор>

<список переменных-записей>::=<переменная-запись>
   {,<переменная-запись>}
```

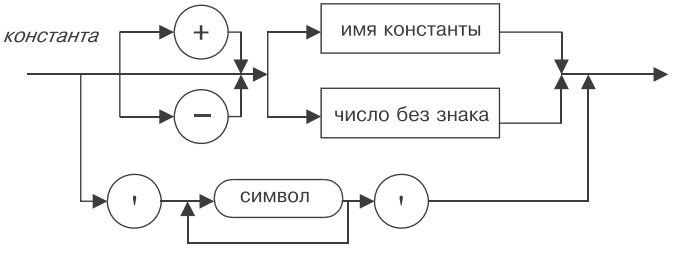
Синтаксические диаграммы



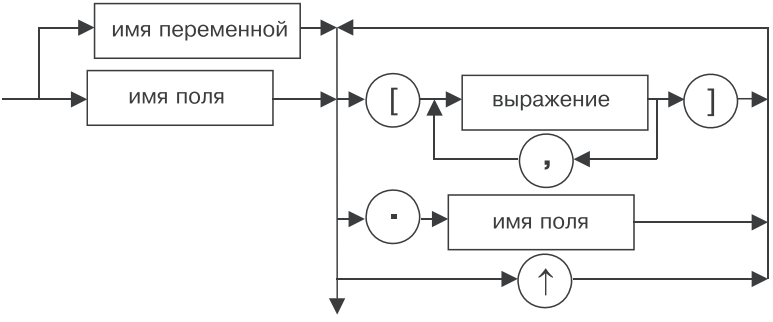




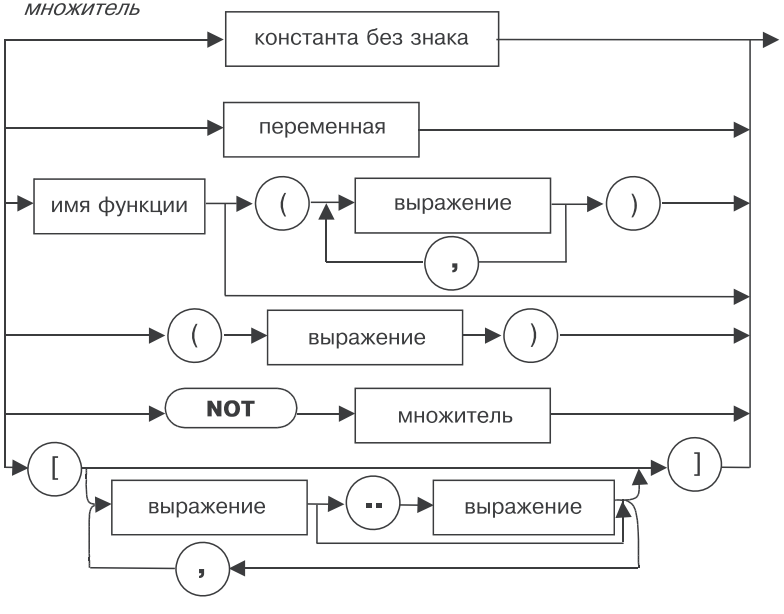




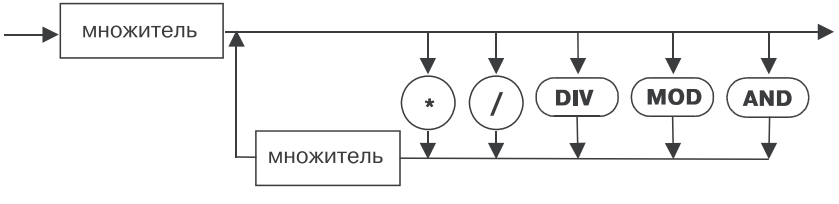
переменная



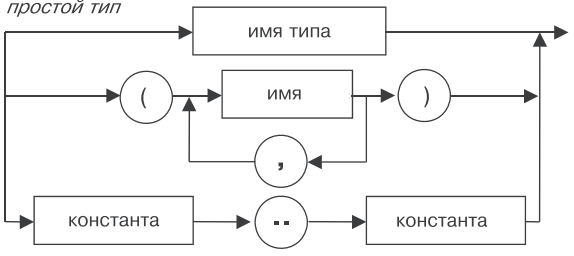
множитель



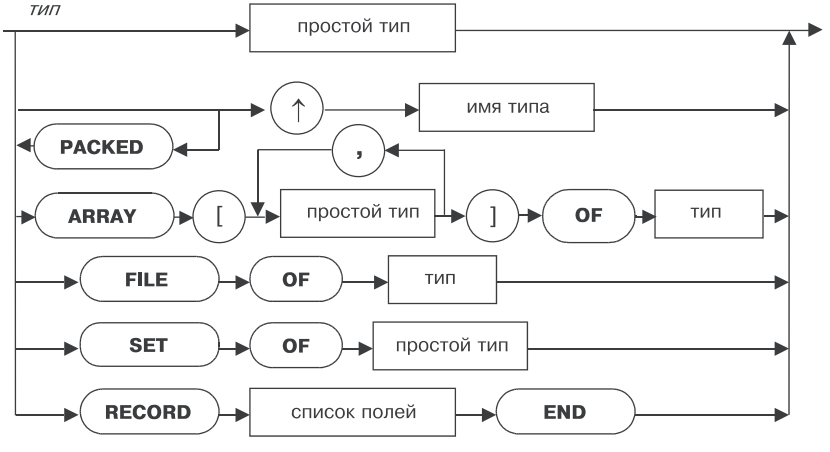
слагаемое



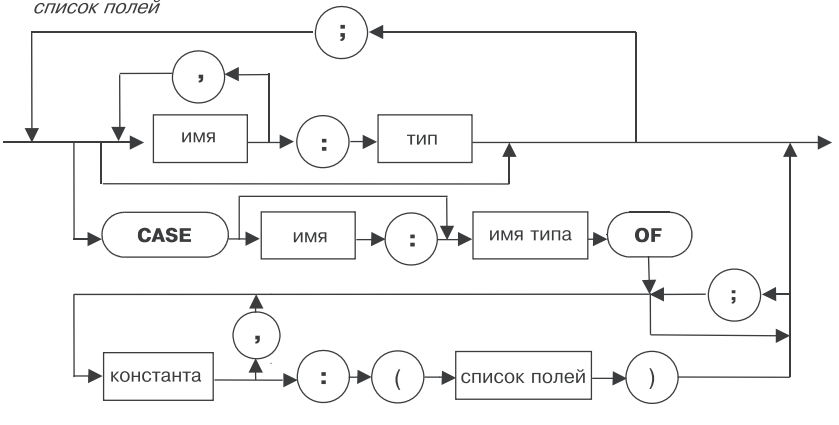
простой тип



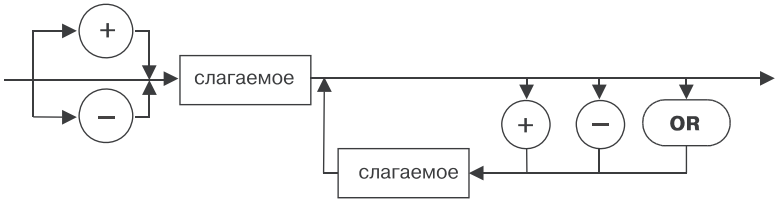
тип



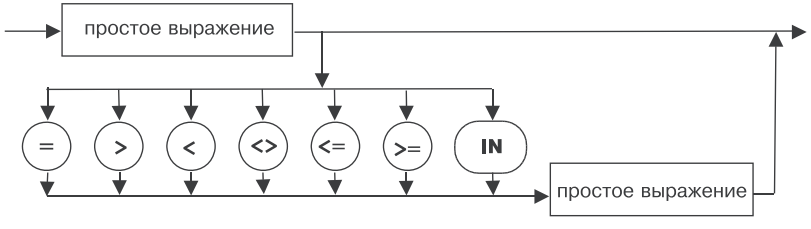
список полей



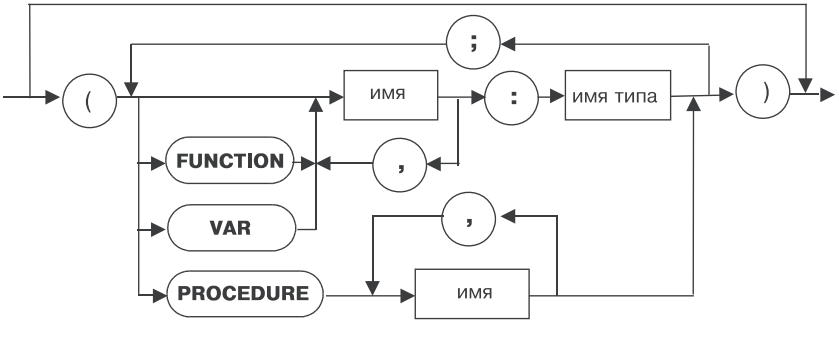
простое выражение



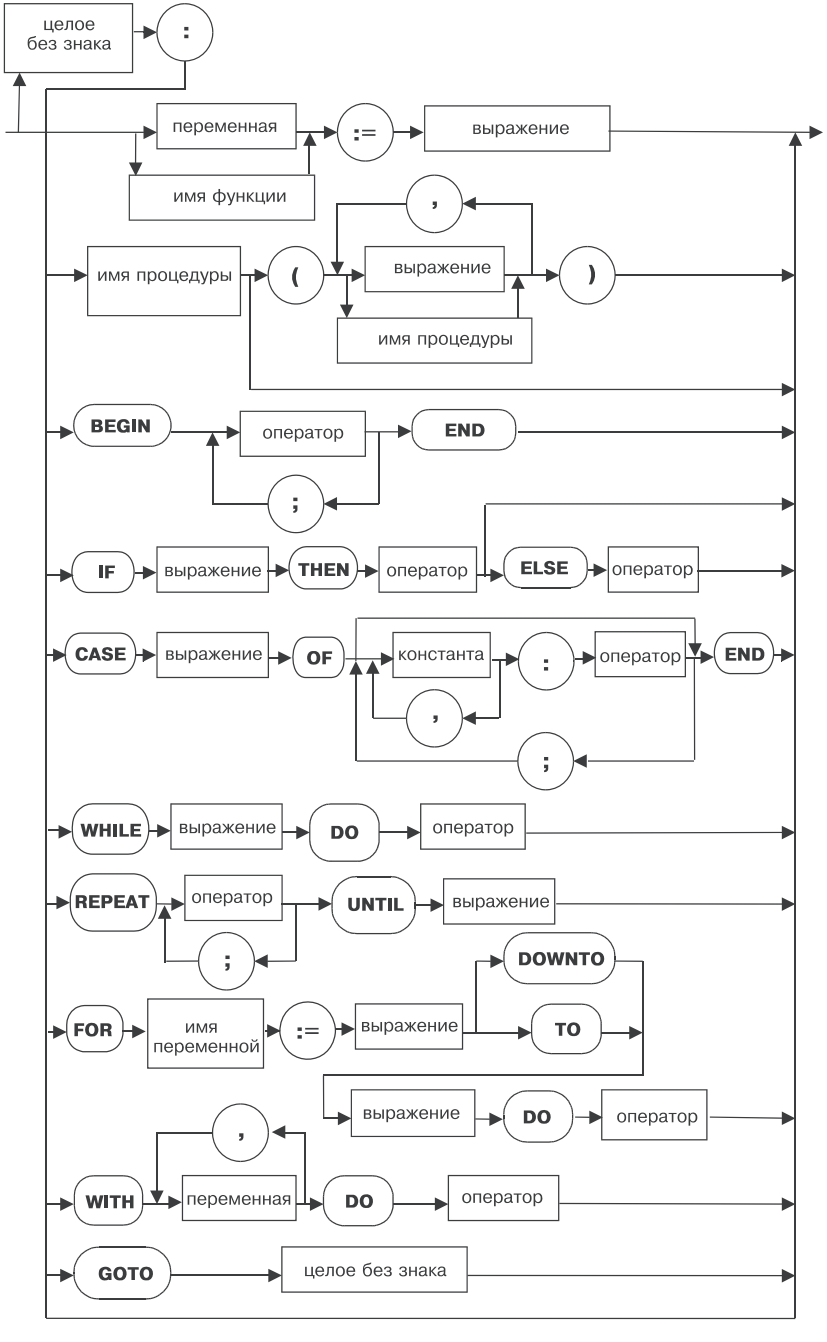
выражение



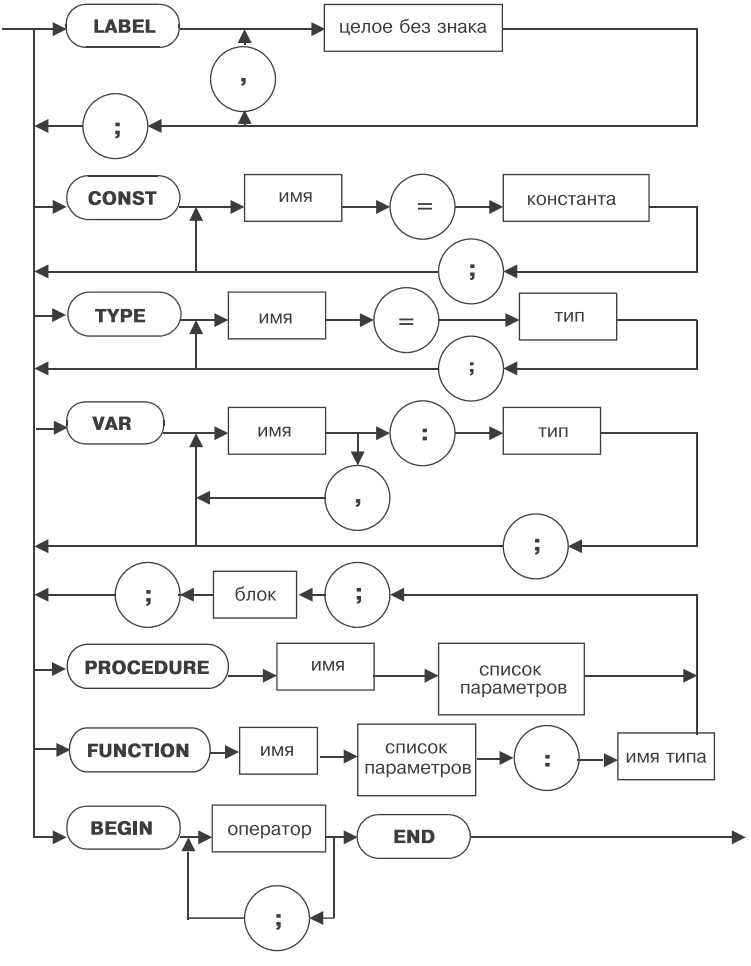
список параметров



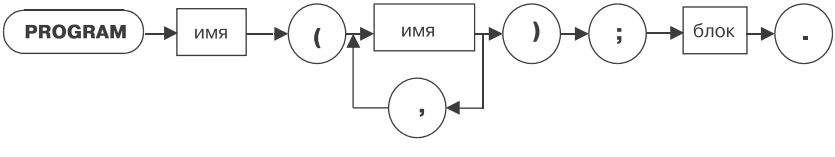
оператор


целое без знака
:
переменная
:=
выражение
имя функции
,
имя процедуры
(
выражение
)
имя процедуры
BEGIN
оператор
END
;
IF
выражение
THEN
оператор
ELSE
оператор
CASE
выражение
OF
константа
:
оператор
END
,
;
WHILE
выражение
DO
оператор
REPEAT
оператор
UNTIL
выражение
;
DOWNTO
FOR
имя переменной
:=
выражение
TO
выражение
DO
оператор
,
WITH
переменная
DO
оператор
GOTO
целое без знака

блок



программа



Приложение 2

Сообщения об ошибках Паскаль-компилятора

1 : ошибка в простом типе
2 : должно идти имя
3 : должно быть служебное слово PROGRAM
4 : должен идти символ ‘)’
5 : должен идти символ ‘:’
6 : запрещенный символ
7 : ошибка в списке параметров
8 : должно идти OF
9 : должен идти символ ‘(‘
10 : ошибка в типе
11 : должен идти символ ‘[‘
12 : должен идти символ ‘]’
13 : должно идти слово END
14 : должен идти символ ‘;’
15 : должно идти целое
16 : должен идти символ ‘=’
17 : должно идти слово BEGIN
18 : ошибка в разделе описаний
19 : ошибка в списке полей
20 : должен идти символ ‘,’
50 : ошибка в константе
51 : должен идти символ ‘:=’
52 : должно идти слово THEN
53 : должно идти слово UNTIL
54 : должно идти слово DO
55 : должно идти слово TO или DOWNTO
56 : должно идти слово IF
58 : должно идти слово TO или DOWNTO
61 : должен идти символ ‘.’
74 : должен идти символ ‘..’
75 : ошибка в символьной константе
76 : слишком длинная строковая константа
86 : комментарий не закрыт
100 : использование имени не соответствует описанию
101 : имя описано повторно
102 : нижняя граница превосходит верхнюю

104 : имя не описано
105 : недопустимое рекурсивное определение
108 : файл здесь использовать нельзя
109 : тип не должен быть REAL
111 : несовместимость с типом дискриминанта
112 : недопустимый ограниченный тип
114 : тип основания не должен быть REAL или INTEGER
115 : файл должен быть текстовым
116 : ошибка в типе параметра стандартной процедуры
117 : неподходящее опережающее описание
118 : недопустимый тип признака вариантной части записи
119 : опережающее описание: повторение списка параметров не допускается
120 : тип результата функции должен быть скалярным, ссылочным или ограниченным
121 : параметр-значение не может быть файлом
122 : опережающее описание функции: повторять тип результата нельзя
123 : в описании функции пропущен тип результата
124 : F-формат только для REAL
125 : ошибка в типе параметра стандартной функции
126 : число параметров не согласуется с описанием
127 : недопустимая подстановка параметров
128 : тип результата функции не соответствует описанию
130 : выражение не относится к множественному типу
131 : элементы множества не должны выходить из диапазона 0 .. 255
135 : тип операнда должен быть BOOLEAN
137 : недопустимые типы элементов множества
138 : переменная не есть массив
139 : тип индекса не соответствует описанию
140 : переменная не есть запись
141 : переменная должна быть файлом или ссылкой
142 : недопустимая подстановка параметров
143 : недопустимый тип параметра цикла
144 : недопустимый тип выражения
145 : конфликт типов
147 : тип метки не совпадает с типом выбирающего выражения
149 : тип индекса не может быть REAL или INTEGER
152 : в этой записи нет такого поля
156 : метка варианта определяется несколько раз
165 : метка определяется несколько раз
166 : метка описывается несколько раз
167 : неописанная метка

168 : неопределенная метка
169 : ошибка в основании множества (в базовом типе)
170 : тип не может быть упакован
177 : здесь не допускается присваивание имени функции
182 : типы не совместны
183 : запрещенная в данном контексте операция
184 : элемент этого типа не может иметь знак
186 : несоответствие типов для операции отношения
189 : конфликт типов параметров
190 : повторное опережающее описание
191 : ошибка в конструкторе множества
193 : лишний индекс для доступа к элементу массива
194 : указано слишком мало индексов для доступа к элементу массива
195 : выбирающая константа вне границ описанного диапазона
196 : недопустимый тип выбирающей константы
197 : параметры процедуры (функции) должны быть параметрами-значениями
198 : несоответствие количества параметров параметра-процедуры (функции)
199 : несоответствие типов параметров параметра-процедуры (функции)
200 : тип парамера-функции не соответствует описанию
201 : ошибка в вещественной константе: должна идти цифра
203 : целая константа превышает предел
204 : деление на нуль
206 : слишком маленькая вещественная константа
207 : слишком большая вещественная константа
208 : недопустимые типы операндов операции IN
209 : вторым операндом IN должно быть множество
210 : операнды AND, NOT, OR должны быть булевыми
211 : недопустимые типы операндов операции + или —
212 : операнды DIV и MOD должны быть целыми
213 : недопустимые типы операндов операции *
214 : недопустимые типы операндов операции /
215 : первым операндом IN должно быть выражение базового типа множества
216 : опережающее описание есть, полного нет
305 : индексное значение выходит за границы
306 : присваиваемое значение выходит за границы
307 : выражение для элемента множества выходит за пределы
308 : выражение выходит за допустимые пределы

Приложение 3

Коды команд для *C*- и *P*-регистров

Наименование операции	Код операции
Вычитание обратное	001
Вычитание модулей обратное	002
Вычитание двоичных чисел обратное	003
Умножение	005
Деление	006
Деление обратное	007
Целая часть в формате числа с плавающей точкой	011
Целая часть в словном формате целого	012
Преобразование целого числа в число с плавающей точкой	013
Сдвиг влево по константе	014
Сдвиг вправо по константе	015
Вычитание	021
Вычитание модулей	022
Вычитание двоичных чисел	023
Сложение	024
Сложение двоичных чисел	025
Сложение модулей	026
Сложение двоичных чисел циклическое	027
Вычитание целых чисел обратное	030
Вычитание целых чисел	031
Умножение целых чисел	032
Сложение целых чисел	033
Пересылка константы в С-регистр	035
Пересылка из *C*- в *P*-регистр	036
Пересылка из *P*- в *C*-регистр	037
Логическое умножение на логическое значение (обратное)	042
Логическое умножение на отрицание логического значения (обратное)	043
Сдвиг влево обратный	044
Сдвиг вправо обратный	045

Наименование операции	Код операции
Неэквивалентность	046
Логическое умножение	047
Выработка логического значения сравнением с нулевым битовым набором	050
Выработка логического значения сравнением с ненулевым битовым набором	051
Выработка логического значения по условию «равно 0»	052
Выработка логического значения по условию «не равно 0»	053
Выработка логического значения по условию «больше или равно 0»	054
Выработка логического значения по условию «меньше 0»	055
Выработка логического значения по условию «меньше или равно 0»	056
Выработка логического значения по условию «больше 0»	057
Логическое умножение на логическое значение	062
Логическое умножение на отрицание логического значения	063
Сдвиг влево	064
Сдвиг вправо	065
Логическое сложение	066
Отрицание	072
Логическое отрицание	073
Изменение знака числа	074
Модуль числа	075
Вычисление номера старшей единицы	076
Вычисление числа единиц	077

Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 11-й для операционных систем Windows, Mac OS, Android, iOS, Windows Phone и BlackBerry; экран 10"

Учебное электронное издание

Залогова Любовь Алексеевна

РАЗРАБОТКА ПАСКАЛЬ-КОМПИЛЯТОРА

Ведущий редактор *Д. Ю. Усенков*
Художественный редактор *Н. А. Лозинская*
Корректор *Е. Н. Клитина*
Компьютерная верстка: *С. А. Янковая*

Подписано к использованию 06.10.14.

Издательство «БИНОМ. Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499)157-5272
e-mail: binom@Lbz.ru
http://www.Lbz.ru, http://e-umk.Lbz.ru, http://metodist.Lbz.ru