Министерство образования и науки Российской Федерации

ГОУ ВПО "Тамбовский государственный технический университет"

И.Л. КОРОБОВА, И.А. ДЬЯКОВ, Ю.В. ЛИТОВКА

ОСНОВЫ РАЗРАБОТКИ ТРАНСЛЯТОРОВ В САПР

Учебное пособие
по дисциплине "Лингвистическое и программное обеспечение САПР"
для студентов 2 курса дневного отделения
специальности 230104

ТАМБОВ
ИЗДАТЕЛЬСТВО Тгту
2007

УДК (681:004.4'42) (075)
ББК ←973-018-5-05я73
К68

Рецензенты:

Кандидат технических наук, доцент кафедры АПТО ТГТУ
М.Н. Краснянский

Кандидат технических наук, доцент кафедры
компьютерного и математического моделирования
ТГУ им. Г.Р. Державина
В.П. Дудаков

Коробова, И.Л.
К68 Основы разработки трансляторов в САПР : учебное пособие / И.Л. Коробова, И.А. Дьяков, Ю.В. Литовка. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2007. – 80 с. – 100 экз. – ISBN 5-8265-0591-5 (978-5-8265-0591-5).

Рассматриваются общие сведения по структуре и составу трансляторов, даны рекомендации по разработке транслирующих программ, рассматриваются различные методы синтаксического анализа и генерации кода, приведены вопросы для самостоятельного изучения.

Предназначено для студентов 2 курса дневного отделения специальности 230104.

УДК (681:004.4'42) (075)
ББК ←973-018-5-05я73

ISBN 5-8265-0591-5
(978-5-8265-

© ГОУ ВПО "Тамбовский государственный технический университет" (ТГТУ), 2007

Учебное издание

КОРОБОВА Ирина Львовна,
ДЬЯКОВ Игорь Алексеевич,
ЛИТОВКА Юрий Владимирович

ОСНОВЫ РАЗРАБОТКИ
ТРАНСЛЯТОРОВ В САПР

Учебное пособие

Редактор О.М. Я р ц е в а
Инженер по компьютерному макетированию Е.В. К о р а б л е в а

Подписано в печать 18.04.2007
Формат 60 × 84/16. 4,65 усл. печ. л. Тираж 100 экз. Заказ № 294

Издательско-полиграфический центр
Тамбовского государственного технического университета
392000, Тамбов, Советская, 106, к. 14

ВВЕДЕНИЕ

В состав любой вычислительной системы может входить комплекс программ, которые называются трансляторами. Транслятор обеспечивает автоматический перевод программ с алгоритмического языка в машинные коды.

По функциональному назначению трансляторы делятся на компиляторы (перевод программ на языке высокого уровня в машинные коды без выполнения), интерпретаторы (перевод каждой конструкции алгоритмического языка в машинные коды с одновременным выполнением) и ассемблеры (перевод программы с языка низкого уровня в машинные коды).

Более подробно остановимся на компиляторах. Компилятор – это не что иное, как программа, написанная на некотором языке, для которой входной информацией служит исходная программа, а результатом является эквивалентная ей объектная программа. Раньше компиляторы писались на автокоде. Часто это был единственно доступный язык. Однако сейчас существует тенденция писать компиляторы на языках высокого уровня, поскольку при этом уменьшается время, затрачиваемое на программирование и отладку, а также обеспечивается удобочитаемость компилятора, когда работа над ним завершена.

Компиляторам присущ ряд общих черт, что упрощает процесс создания компилирующих программ. Наша цель состоит в том, чтобы описать известные уже модельные представления структуры компиляторов и показать, как с их помощью создается работоспособная компилирующая программа.

Компилятор должен выполнить анализ исходной программы и синтез объектного кода. В соответствии с этим любой компилятор включает три основные части: лексический анализатор, синтаксический анализатор и генератор кода.

Взаимодействие между компонентами компилятора может осуществляться разнообразными способами.

В настоящей работе рассматриваются основные подходы к созданию транслирующих программ. Приведенные подходы будут полезны для студентов 2-го курса специальности 230104 – "САПР" при выполнении лабораторных и курсовой работы по дисциплине "Лингвистическое и программное обеспечение САПР".

1.1. ЛЕКСИЧЕСКИЙ АНАЛИЗ

На вход компилятора, а следовательно, и лексического анализатора поступает цепочка символов некоторого алфавита. Работа лексического анализатора заключается в том, чтобы сгруппировать определенные символы в единые синтаксические объекты, называемые *лексемами*. Какие объекты считать лексемами, зависит от определения языка. Кроме терминальных символов (+, –, /, *, (,)), которые сами по себе являются лексемами, в программе некоторые комбинации символов часто рассматриваются как единые объекты. Среди типичных примеров можно указать следующие.

- В некоторых языках цепочка, состоящая из одного или более пробелов, обычно рассматривается как один пробел.
- В языках программирования есть ключевые слова, такие, как begin, end, to, do, integer и другие, каждое из которых считается одним символом.
- Каждая цепочка, представляющая цифровую константу, рассматривается как один элемент текста.
- Идентификаторы, используемые имена переменных, функций, процедур, меток и т.п., также считаются лексическими единицами алгоритмического языка.

Для программы на рис. 1 будем считать лексемами терминальные символы, относящиеся к ключевым словам, знакам операций, разделителям (program, var, begin, integer, end, (,), :=, +, –, *, div, read, for, write, to, do, :, ; , .). Кроме того, возможны лексемы – идентификаторы и константы.

```
program primer;
var sum, a, rez, i: integer;
begin
sum:=0;
for i:=1 to 100 do begin
read(a);
sum:=sum+a
end;
rez:=sum div100–a*a;
write(rez,sum)
end.
```

Рис. 1

Итак, лексический анализатор должен исходный текст программы (рис. 1) представить в виде последовательности лексем. Для эффективности последующих действий каждая лексема обычно представляется некоторым кодом фиксированной длины (например, целым числом), а не в виде строки символов переменной длины.

Для вышеприведенного примера можно составить кодировочную таблицу (табл. 1). Если распознанная лексема является ключевым словом, разделителем или знаком операции, такая схема кодирования дает всю необходимую информацию.

В случае идентификатора или константы необходимы дополнительные данные (в простейшем случае – тип и указание на адрес ячейки памяти, где они хранятся). Обычно эти данные находятся в таблицах символов, и в качестве дополнительной информации для лексемы типа идентификатор или константа может служить указатель на соответствующий элемент таблицы.

Таблица 1

Лексема	Код
program	1
var	2
begin	3
end	4
integer	5
for	6
read	7
write	8
to	9
do	10
.	11
;	12
:	13
,	14
:=	15
+	16
–	17
*	18
div	19
(	20
)	21
ид	22
конст	23

Таким образом, результат обработки лексическим анализатором обрабатываемой программы можно представить последовательностью лексем (табл. 2). Здесь в качестве дополнительных данных для констант используется значение самой константы, а для идентификатора – его номер в таблице символов.

Таблица 2

Строка	Код лексемы	Дополнительные данные
1	1	
	22	1
	12	
2	2	
	22	2
	14	
	22	3
	14	
	22	4

	14	
	22	5
	13	
	5	
	12	
3	3	
4	22	2
	15	
	12	
5	6	
	22	5
	15	
	23	# 1
	9	
	23	# 100
	10	
6	3	
7	7	
	20	
	22	3
	21	
	12	
8	22	2
	15	
	22	2
	16	
	22	3
9	4	
	12	
10	22	4
	15	
	22	2
	19	
	23	#100
1	2	3
	22	3
	18	
	22	3

	12	
11	8	
	20	
	22	4
	14	
	22	2
	21	
12	4	
	11	

1.2. СИНТАКСИЧЕСКИЙ АНАЛИЗ

Синтаксический анализ – второй этап компиляции. Во время этого этапа предложения программы распознаются как языковые конструкции используемой грамматики. Для того чтобы выяснить, принадлежит ли предложение языку, необходимо построить алгоритм, который для любого предложения, допустимого грамматикой, давал бы последовательность выводов этой цепочки к начальному символу грамматики. Мы можем рассматривать этот процесс как построение дерева грамматического разбора для транслируемых предложений. Различают две категории алгоритмов разбора: нисходящий (сверху вниз) и восходящий (снизу вверх). Эти термины соответствуют способу построения синтаксических деревьев. Рассмотрим для примера предложение 35 в грамматике целых чисел:

$$N \rightarrow B/NB; \quad B \rightarrow 0/1/2/3/4/5/6/7/8/9.$$

При нисходящем разборе дерево строится от корня (начального символа) вниз к концевым узлам (рис. 2).

$$N \rightarrow NB \rightarrow N5 \rightarrow B5 \rightarrow 35$$

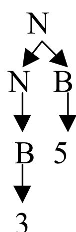


Рис. 2

Восходящий разбор состоит в том, что, отправляясь от заданной цепочки, пытаются привести ее к начальному символу (рис. 3).

$$35 \rightarrow B5 \rightarrow N5 \rightarrow NB \rightarrow N$$

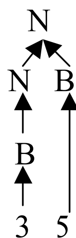


Рис. 3

Разработано множество методов синтаксического анализа. В лабораторных работах рассматриваются два метода: нисходящий и восходящий.

1.2.1. Метод рекурсивного спуска

Процесс грамматического разбора для этого метода состоит из отдельных процедур для каждого нетерминального символа, определенного в грамматике. Каждая такая процедура старается во входном потоке найти подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов. Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает работу и передает в вызывающую ее программу признак успешного выполнения. Затем рассматривается следующая лексема, идущая за распознанной подстрокой. Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком неудачи или же вызывает процедуру диагностического сообщения.

Рассмотрим в качестве примера правило грамматики:

$$\langle\text{ввод}\rangle \rightarrow \text{read} \ (\langle\text{список переменных}\rangle)$$

Процедура метода рекурсивного спуска, соответствующая нетерминальному символу <ввод>, прежде всего исследует две последовательные лексемы "read" и "(". В случае совпадения эта процедура вызывает другую процедуру, соответствующую нетерминальному символу <список переменных>. Если эта процедура закончится успешно, то процедура <ввод> сравнивает следующую лексему с ")". Если все эти проверки окажутся успешными, то процедура <ввод> завершается с признаком успеха и устанавливает указатель текущей лексемы на лексему, следующую за ")".

Еще пример. Процедура, соответствующая нетерминальному символу <оператор>, анализирует очередную лексему для того, чтобы выбрать одну из четырех альтернатив:

$$\langle\text{оператор}\rangle \rightarrow \langle\text{присваивание}\rangle/\langle\text{ввод}\rangle/\langle\text{вывод}\rangle/\langle\text{цикл}\rangle$$

Если это лексема read, то вызывается процедура <ввод>. Если это лексема, соответствующая символу идентификатор, то вызывается процедура <присваивание>, поскольку это единственная альтернатива, которая может начинаться с лексемы идентификатор, и т.д.

Но если мы попытаемся написать полный набор процедур для грамматики, то столкнемся со следующей трудностью – процедура для нетерминала <список переменных> будет не в состоянии выбрать одну из двух альтернатив, поскольку обе альтернативы: ид и <список переменных> могут начинаться с лексемы ид.

$$\langle\text{список переменных}\rangle \rightarrow \text{ид}/\langle\text{список переменных}\rangle, \text{ид}$$

Тут скрыта и более существенная трудность. Если процедура каким-либо образом решит попробовать альтернативу <список переменных>/ид, то она немедленно вызовет рекурсивно саму себя для поиска нетерминального символа <список переменных>. Это приведет еще к одному рекурсивному вызову и т.д., в результате чего образуется бесконечная цепочка рекурсивных вызовов. Те же проблемы возникнут и для некоторых других правил грамматики (<раздел переменных>, <раздел операторов>, <арифметическое выражение>, <слагаемое>). Как избежать такой рекурсии? Для этого применяют другую запись грамматики. Например:

$$\langle\text{список переменных}\rangle \rightarrow \text{ид} \ \{, \text{ид}\}$$

Эта запись, являющаяся широко принятым расширением БНФ, означает, что конструкция, заключенная в фигурные скобки, может быть либо опущена, либо повторяться один или более число раз. Таким образом, это правило определяет нетерминальный символ <список переменных> как состоящий из единственной лексемы ид или же из произвольного числа следующих друг за другом лексем ид, разделенных запятой. Это, бесспорно, эквивалентно ранее принятому правилу. В соответствии с этим новым определением процедура <список переменных> сначала ищет лексему ид, а затем продолжает сканировать входной текст до тех пор, пока следующая пара лексем не совпадет с запятой и ид. Такая запись устраняет проблему рекурсии, а также решает вопрос выбора из двух альтернатив.

Грамматика языка, к которому принадлежит предложение, представленное на рис. 1, имеет вид:

<программа> → <имя программы> var <раздел
переменных> begin <раздел операторов> end.
<имя программы> → ид
<раздел переменных> → <список переменных>:<тип>
<список переменных> → ид {, ид}
<тип> → integer
<оператор> → <присваивание> / <ввод> /<вывод> /<цикл>
<присваивание> → ид := <арифметическое выражение>
<арифметическое выражение> → <слагаемое> {+<слагаемое>} {–<слагаемое>}
<слагаемое> → <значение> {*<значение>} {div <значение>}
<ввод> → read (<список переменных>)
<вывод> → write (<список переменных>)
<цикл> → for <выражение цикла> to <тело цикла>
<выражение цикла> → ид :=<арифметическое выражение>
do <арифметическое выражение>
<тело цикла> → <оператор>/ begin <раздел операторов> end

Приведем примеры алгоритмов синтаксического анализа методом рекурсивного спуска для некоторых предложений исходной программы с использованием приведенной грамматики.

Имеем предложение исходной программы: read (a).

Тогда процедура разбора этого предложения может иметь вид:

```
procedure <ввод>;
   begin
BP := false;
 if t = read then
   begin
    перейти к следующей лексеме;
    if  t = (  then   begin
       перейти к следующей лексеме;
        if <список переменных> закончилась успешно
           then
              if t  = )  then   begin
                BP := true;
                    перейти к следующей лексеме;
                      end; {if )}
                     end; {if (}
             end; {if read}
             if BP = true then успешное завершение
                    else неудачное завершение;
          end;
```

В приведенной процедуре BP – вспомогательная переменная, а t – переменная, определяющая тип лексемы. Процедура, соответствующая нетерминальному символу <ввод>, вызывает процедуру <список переменных>:

```
procedure <список переменных>;
    begin
   BP := false;
   if t=ид then
      begin
```

```
BP := true;
перейти к следующей лексеме;
while ( t = , )  and  ( BP = true ) do
     begin
         перейти к следующей лексеме;
         if t = ид  then перейти к следующей
         лексеме
                    else BP := false;
     end; {while}
if BP = true then успешное завершение
                 else неудачное завершение;
end;
```

На рис. 4 графически представлен процесс грамматического разбора методом рекурсивного спуска для предложения read. На фрагменте а) изображен вызов процедуры <ввод>, которая обнаружила лексемы read и) во входном потоке (штриховая линия). На фрагменте б) процедура <ввод> вызывает процедуру <список переменных> (сплошная линия), которая обработала лексему ид. На фрагменте в) процедура <список переменных> закончила работу, передала управление процедуре <ввод> с признаком успешного завершения; процедура <ввод> обработала входную лексему). На этом анализ входного предложения завершен.

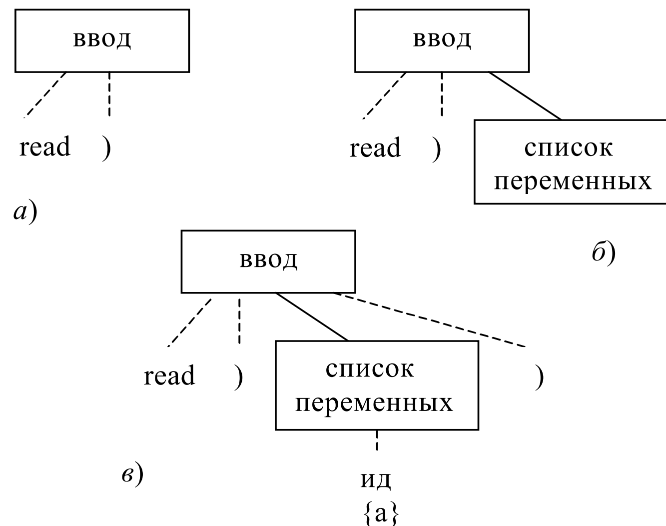


Рис. 4

Приведем еще один пример. Имеем предложение из исходной программы:

REZ := SUM DIV 100 – A * A

Представим алгоритмы разбора этого предложения методом рекурсивного спуска:

```
procedure <присваивание>
        begin
      BP := false;
      if  t  =  ид   then
          begin
              перейти к следующей лексеме;
               if t  =   :=   then
                   begin
               перейти к следующей лексеме;
                      if <арифметическое выражение>
                      завершилось успешно then
```

```
                                        BP := true;
                              end; {if :=}
                       end;  {if ид.}
                  if BP = true then успешное завершение
                          else неудачное завершение;
         end;
```

Процедура присваивание в процессе работы вызывает процедуру <арифметическое выражение>:

```
procedure <арифметическое выражение>;
   begin
  BP:=false;
  if  <слагаемое>  завершилось успешно then
     begin
          BP:=true;
          while (t = +  или  t = – ) and ( BP=true ) do
              begin
                   Перейти к следующей лексеме;
                   if  <слагаемое>  завершилось неудачно
                   then
                      BP:=false;
               end; {while}
     end; {if слагаемое}
  if  BP = true then успешное завершение
     else неудачное завершение;
end;
```

Процедура <арифметическое выражение>, в соответствии с грамматикой, вызывает процедуру <слагаемое>:

```
procedure <слагаемое> ;
   begin
       BP:=false;
  if <значение>  завершилось успешно then
     begin
          BP:=true;
          while (t = *  или  t = div ) and ( BP=true ) do
              begin
                   Перейти к следующей лексеме;
                   if <значение> завершилось неудачно
                   then
                      BP:=false;
              end; {while}
     end; {if значение}
  if  BP=true then успешное завершение
              else неудачное завершение;
   end;
```

И наконец, процедура <слагаемое> вызывает процедуру <значение>, которая распознает переменные, константы или вызывает процедуру <арифметическое выражение>. Алгоритм процедуры <значение> имеет вид:

```
procedure значение;
```

```
 begin
BP:=false;
if t =  ид  или  t = конст    then
   begin
         BP:=true;
         Перейти к следующей лексеме;
   end {if ид или конст}
          else
                if t = (   then        begin
              Перейти к следующей лексеме;
            if <арифметическое выражение>  завершилось  успешно  then
                if  t = )   then
                    begin
                       BP:=true;
                    Перейти к следующей лексеме;
                      end; {if )}
             end; {if (}
if BP = true then успешное завершение
                      else неудачное завершение;
 end;
```

Графически разбор этого предложения методом рекурсивного спуска выглядит так:

1. Вызывается процедура <присваивание>, которая обнаружила лексемы ид и := во входном потоке (рис. 5, *а*).
2. Процедура < присваивание > вызывает процедуру <арифметическое выражение> (рис. 5, *б*)
3. Процедура <арифметическое выражение> вызывает процедуру <слагаемое> (рис. 6, *а*).
4. Процедура <слагаемое>. вызывает процедуру <значение>, которая обнаруживает лексему ид. Управление возвращается в процедуру <слагаемое> (рис. 6, *б*).
5. Процедура <слагаемое> обнаруживает лексему div и вызывает процедуру <значение>, которая обнаруживает лексему конст. и передает управление в процедуру <слагаемое>, которая передает управление в процедуру <арифметическое выражение> (рис. 7, *а*).

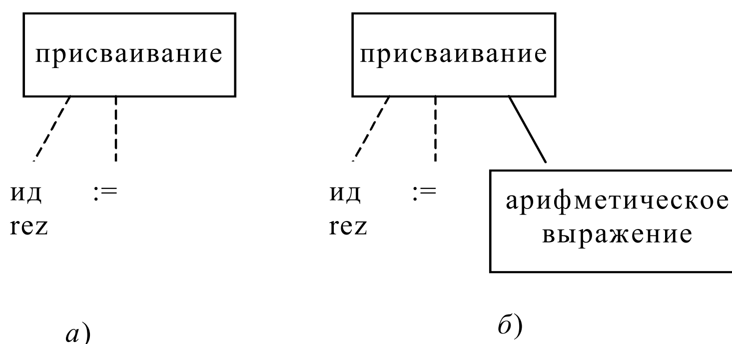
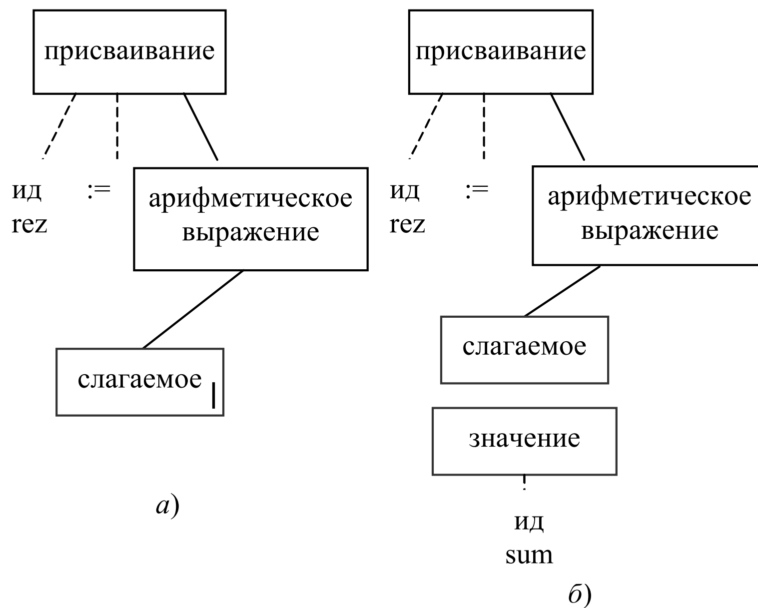


Рис. 5

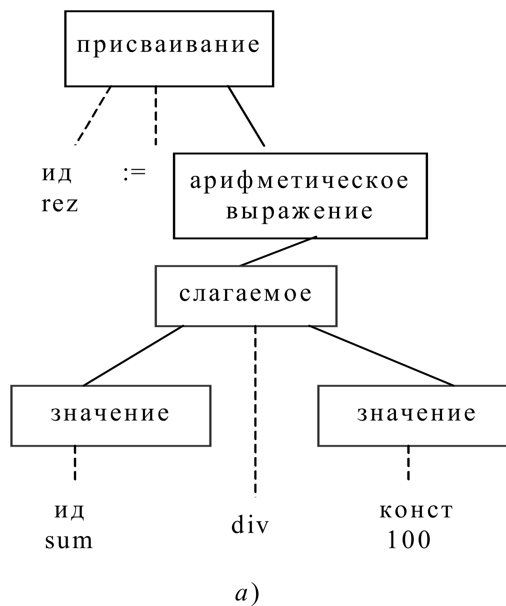


а) *б*)

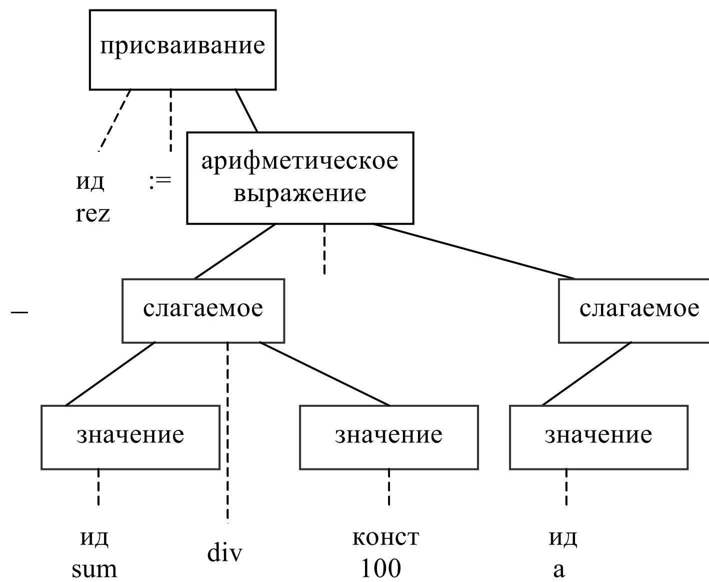
Рис. 6

6. Процедура <арифметическое выражение> распознает лексему –, затем вызывает процедуру <слагаемое>, которая вызывает процедуру <значение>, которая распознает лексему ид и передает управление в процедуру <слагаемое> (рис. 7, *б*).

7. Процедура <слагаемое> распознает лексему *, затем вызывает процедуру <значение>, которая распознает лексему ид. Следующая лексема читается в процедуре <значение>. Эта лексема не относится к данному предложению. Управление передается в процедуру <слагаемое>, которая формирует признак успешного завершения и передает управление в процедуру <арифметическое выражение>. Эта процедура, в свою очередь, успешно заканчивается и передает управление в процедуру <присваивание>, которая формирует признак успешного завершения. На этом разбор этого предложения заканчивается (рис. 8).



а)



б)

Рис. 7

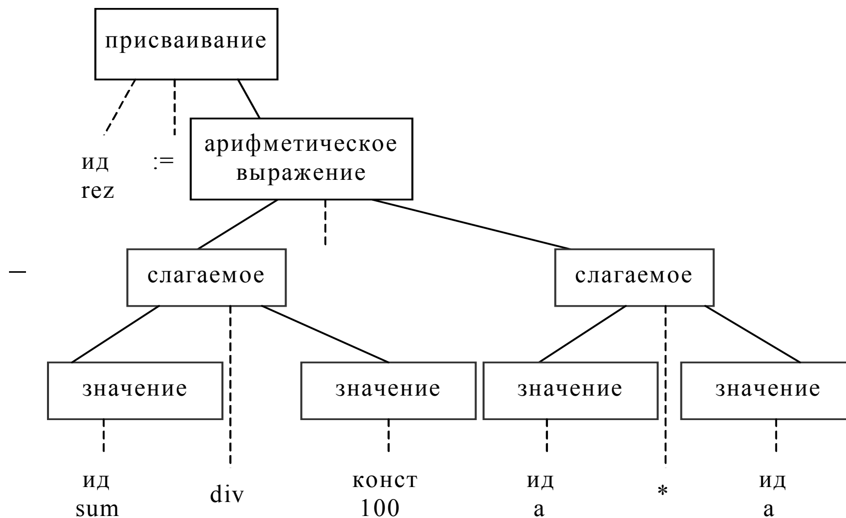


Рис. 8

Мы привели примеры грамматического разбора отдельных предложений методом рекурсивного спуска. Однако этот метод применим и ко всей программе в целом. В этом случае для осуществления синтаксического анализа следует просто обратиться к процедуре, соответствующей нетерминальному символу <программа>. В результате работы этой процедуры будет построено дерево грамматического разбора для всей программы.

1.2.2. Метод операторного предшествования

Этот метод относится к восходящим (метод снизу вверх), которые начинают разбор с конечных узлов грамматического дерева и пытаются объединить их построением узлов все более и более высокого уровня до тех пор, пока не будет достигнут корень дерева. Метод операторного предшествования основан на анализе пар последовательно расположенных операторов исходной программы и решением вопроса о том, какой из них должен выполняться первым. Рассмотрим, например, арифметическое выражение

$$A + B * C - B.$$

В соответствии с обычными правилами арифметики умножение и деление осуществляются до сложения и вычитания. Можно сказать, что умножение и деление имеют более высокий уровень предшествования, чем сложение и вычитание. При анализе первых двух операторов (+, *) выяснится, что оператор + имеет более низкий уровень предшествования, чем оператор *. Часто это записывают следующим образом

+ <• *

Аналогично для следующей пары операторов (* и –) оператор * имеет более высокий уровень предшествования, чем оператор –. Мы можем записать это в виде

*•>–

Метод операторного предшествования использует подобные отношения между операторами для управления процессом грамматического разбора. В частности, для рассмотренного арифметического выражения мы получили следующие отношения предшествования:

A + B * C – B

<•   •>

Отсюда следует, что подвыражение B * C должно быть вычислено до обработки любых других операторов рассматриваемого выражения. В терминах дерева грамматического разбора это означает, что операция * расположена на более низком уровне узлов дерева, чем операция + или –. Таким образом, рассматриваемый метод грамматического разбора должен распознать конструкцию B * C, интерпретируя ее в терминах заданной грамматики, до анализа соседних термов предложения.

Предшествующее изложение иллюстрирует основную идею, на которой основан метод грамматического разбора, построенный на отношения операторного предшествования. В рамках этого метода предложение сканируется слева направо до тех пор, пока не будет найдено подвыражение, операторы которого имеют более высокий уровень предшествования, чем соседние операторы. Далее это подвыражение распознается в терминах правил вывода используемой грамматики. Этот процесс продолжается до тех пор, пока не достигнут корень дерева, что и будет означать окончание процесса грамматического разбора. Далее мы рассмотрим приложение описанного подхода к нашему примеру программы (рис. 1). Грамматика этого предложения имеет вид:

<программа>→<имя программы> var <раздел
переменных> begin <раздел операторов> end.
<имя программы>→ид
<раздел переменных>→<список переменных>:<тип>
<список переменных>→ид / <список переменных>, ид
<тип>→integer
<раздел операторов>→<оператор>/<раздел операторов>; <оператор>
<оператор>→<присваивание>/<ввод>/<вывод>/<цикл>
<присваивание>→ид := <арифметическое выражение>
<арифметическое выражение>→<слагаемое> /<арифметическое выражение>+/–<слагаемое>
<слагаемое> → <значение> >/<слагаемое> */div <значение>
<ввод>→read (<список переменных>)
<вывод>→write (<список переменных>)
<цикл>→for <выражение цикла> to <тело цикла>
<выражение цикла>→ид :=<арифметическое выражение> do <арифметическое выражение>
<тело цикла>→<оператор>/begin <раздел операторов>
end

Первым шагом при разработке процессора грамматического разбора, основанного на методе операторного предшествования, должно быть установление отношений предшествования между операторами

грамматиками. При этом под оператором понимается любой терминальный символ (т.е. любая лексема). Таким образом, мы должны, в частности, установить отношения предшествования между лексемами begin, read, (. Приведем матрицу, которая задает отношения предшествования для нашей грамматики (табл. 4).

Каждая клетка этой матрицы определяет отношение предшествования (если оно существует) между лексемами, соответствующими строке и столбцу, на пересечении которых находится эта клетка. Например, мы видим, что

program = var и

begin <• read

Отношение = означает, что обе лексемы имеют одинаковый уровень предшествования и должны рассматриваться грамматическим процессором в качестве составляющих одной конструкции языка. Обратите внимание, что для отношения предшествования не выполняются некоторые правила, привычные для отношения арифметического порядка. Например,

; •> end,   но   end •> ;

Обратите внимание также на то, что для многих пар лексем отношения предшествования не существует. Это означает, что соответствующие пары лексем не могут находиться рядом ни в каком грамматически правильном предложении. Если подобная комбинация лексем все же встретится в процессе грамматического разбора, то она должна рассматриваться как синтаксическая ошибка.

Для применимости метода операторного предшествования необходимо, чтобы отношения предшествования были заданы одиозначно, например, не должно быть одновременно отношений ; <• begin и ; >• begin. Это требование выполняется для нашей грамматики, однако, несущественные ее изменения могут привести к тому, что некоторые из отношений перестанут быть однозначными и метод операторного предшествования станет не применимым.

Приведем пример разбора с помощью операторного предшествования. Пусть анализируется предложение read из нашей программы. Это предложение анализируется по лексемам слева направо. Для каждой пары соседних операторов определено отношение предшествования

```
begin   read   (     ид.    )   ;
         <•      =    <•.      •>  •>
begin   read   (  <N1>  )    ;
         <•      =      =        •>
begin    <N2>     ;
            <•
```

На первом шаге процесса грамматического разбора выделим фрагмент, ограниченный отношениями <• и •> для распознавания в терминах грамматики. В данном случае этот фрагмент содержит единственную лексему ид. Эта лексема может быть распознана как нетерминал <значение> в соответствии с правилом из грамматики.

Лексемы	var	begin	end	End.	Integer	for	read	write	to	do	;
program	=										
var		=									<
begin			=	=		<	<	<			<
end			>	>							>
integer		>									>
for											

read											
write											
to										>	
do		<	>	>		<	<	<			>
;		>	>	>		<	<	<			>
:					<						
,											
:=			>	>					=		>
+			>	>					>	>	>
-			>	>					>	>	>
*			>	>					>	>	>
div			>	>					>	>	>
(											
)											
ид	>		>	>					>	>	>
конст			>	>					>	>	>

Таблица 4

:	,	:=	+	-	*	div	(	)	ид	конст
									<	
<	<								<	
									<	
									<	
							=			
							=			
			<	<	<	<	<		<	<
									<	
<	<								<	
									<	
			<	<	<	<	<		<	<
			>	>	<	<	<	>	<	<
			>	>	<	<	<	>	<	<
			>	>	>	>	<	>	<	<
			>	>	>	>	<	>	<	<
	<		<	<	<	<	<	=	<	<
			>	>	>	>		>		
>	>	=	>	>	>	>		>		
			>	>	>	>		>		

Однако эта лексема может быть также распознана как нетерминальный символ <список переменных>. Для рассматриваемого метода неважно, какой конкретно нетерминальный символ распознан. Лексема ид интерпретируется просто как некий нетерминальный символ <N1>. Конструкция read(<N1>) интерпретируется как один нетерминальный символ <N2>.

На этом разбор предложения read закончено. Если мы сравним деревья грамматического разбора для этого предложения, то увидим, что они полностью совпадают, за исключением имен нетерминальных символов.

Рассмотрим грамматический разбор для оператора

```
 ;    rez: = sum   div 100   –  A  *  A ;
   <      = <     >     <    >  < > < >
 ;    rez := <N1> div <N2>  –  <N4>  *  <N5>  ;
   <      =     <             >          <          >
 ;    rez := <N3>  –  <N6>  ;
   <      =     <           >
 ;    rez := <N7>  ;
   <      =    >
```

При этом дерево грамматического разбора имеет вид, представленный на рис. 9.

Заметим еще раз, что процесс грамматического разбора начинается слева направо и продолжается на каждом шаге до тех пор, пока не определится очередной фрагмент предложения для грамматического распознавания, т.е. первый фрагмент, ограниченный отношениями <• и •>. Как только подобный фрагмент выделен, он интерпретируется как некоторый очередной нетерминальный символ в соответствии с каким-нибудь правилом грамматики. Этот процесс продолжается до тех пор, пока предложение не будет распознано целиком.

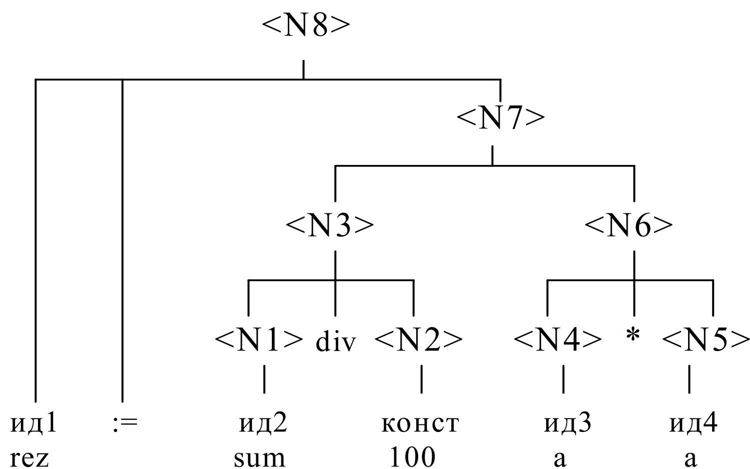


Рис. 9

Обратите внимание, что каждый фрагмент дерева грамматического разбора строится, начиная с конечных узлов вверх, в сторону корня дерева. Отсюда и возник термин восходящий разбор. Если мы рассмотрим дерево грамматического разбора, исходя из грамматики языка (рис. 8), то увидим, что оно несколько отличается от полученного методом операторного предшествования дерева (рис. 9).

Например, идентификатор sum был сначала интерпретирован как <значение>, а потом как <слагаемое>, являющийся одним из операндов операции div. При разборе методом операторного предшествования идентификатор sum был интерпретирован как единственный нетерминал <N1>, который является операндом операции div. Таким образом, <N1> соответствует двум нетерминальным символам <значение> и <слагаемое>. Имеются и другие подобные различия.

Они вытекают из свободы образования имен нетерминальных символов, распознаваемых в рамках метода операторного предшествования. Интерпретация sum сначала как <значение>, а потом как <слагаемое> является просто переименованием нетерминальных символов. Такое переименование необхо-

димо, поскольку в соответствии с грамматикой (правило 8) первым операндом операции умножения должен быть <слагаемое>, а не <значение>. Так как для нашего метода имена нетерминальных символов несущественны, то подобные переименования становятся ненужным. Собственно говоря, три различных имени: <арифметическое выражение>, <слагаемое>, <значение> – были включены в грамматику только как средства описания отношения предшествования между операторами (например, для указания того, что умножение следует выполнять после сложения). Поскольку эта информация содержится в нашей матрице предшествования, то становится ненужным различать эти три имени в процессе грамматического разбора.

Возможный алгоритм метода операторного предшествования приведен на рис. 10.

В данном алгоритме первоначально просматривается цепочка лексем (массив L), устанавливается отношение предшествования между соседними лексемами по таблице отношений предшествования (матрица $M[n \times n]$) до тех пор, пока не встретится отношение '>' (блоки 4–5). После этого возвращаемся по массиву лексем назад, пока между лексемами не встретится отношение '<' (блоки 7–8). Затем лексемы, ограниченные отношениями '< >', записываются во внутреннее представление (блоки 9–10) .

В блоках 12 – 14 элементы массива лексем сдвигаются на длину выведенной цепочки. Так продолжается, пока не распознана вся последовательность лексем.

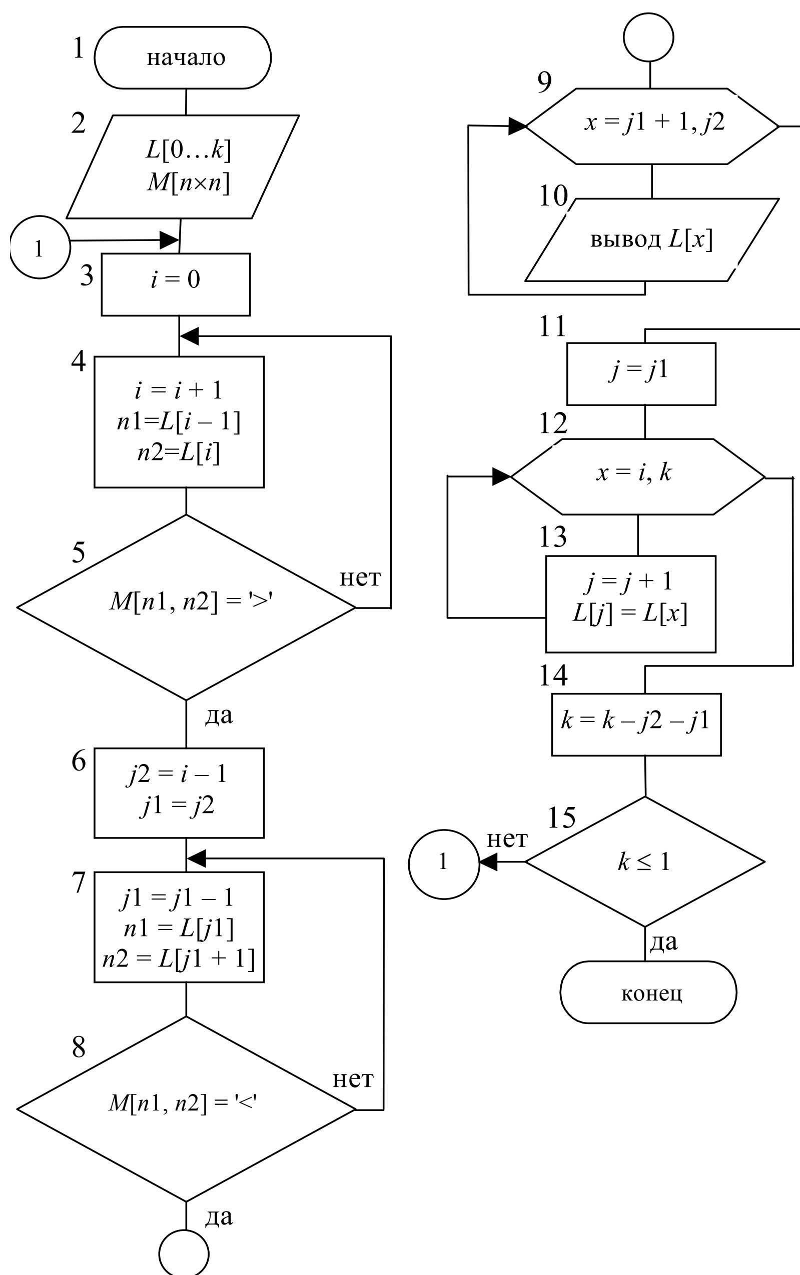


Рис. 10

1.3. ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ПРОГРАММЫ

На выходе синтаксического анализатора формируется программа во внутреннем представлении. Существует несколько различных способов представления программы в некоторой промежуточной форме для анализа и оптимизации кода: последовательность четверок, последовательность троек, постфиксная запись, префиксная запись, синтаксическое дерево.

1.3.1. Последовательность четверок

Каждая четверка записывается в виде

операция op1, op2, результат,

где операция – выполняемая объектным кодом функция; op1, op2 – операнды этой операции; результат определяет, куда должно быть помещено результирующее значение.

Например, предложение исходной программы (рис. 1): sum: = sum + a может быть представлено четверками следующим образом:

```
+      sum ,  a  ,   I1
:=      I1  ,     ,   sum
```

Здесь I1 обозначает промежуточный результат (sum + a), вторая четверка присваивает это значение переменной sum.

Все четверки расположены в том порядке, в котором должны выполняться соответствующие инструкции объектного кода, что существенно облегчает анализ для оптимизации кода. Это означает также, что трансляция в машинные коды будет относительно простой. В табл. 5 представлена последовательность четверок, соответствующая исходной программе.

За операциями read и write следует четверка param, определяющая параметры операций read и write. Четверка param будет, разумеется, при окончательной генерации машинного кода оттранслирована в список параметров. Операция > в четверке 3 сравнивает значение двух своих операндов и осуществляет переход к четверке 9, если первый операнд больше второго. Операция goto в четверке 8 осуществляет безусловный переход к четверке 3.

Таблица 5

	Операция	Операнд 1	Операнд 2	Результат
1	: =	#0		sum
2	:=	#1		I
3	>	I	#100	9)
4	read			
5	param	a		
6	+	sum	a	I1
7	: =	I1		sum
8	goto			3)
9	div	sum	#100	I1
10	*	a	a	I3
11	–	I2	I3	I4
12	:=	I4		rez
13	write			
14	param	rez		
15	param	sum		

Таким образом, последовательность четверок и является результатом работы синтаксического анализатора.

1.3.2. Постфиксная запись

Обычно программа осуществляет те или иные действия над данными. Соответствующие операции программист записывает, используя инфиксную форму записи, в которой знак операции ставится между операндами. Например:

(A + B) * C.

Вычисление такого выражения является непростой задачей. Операцию умножения нельзя выполнить до тех пор, пока не будет прочитан второй операнд C. Если этот операнд сам является сложным выражением, то прежде, чем выполнить умножение, необходимо считать много данных из текста программы.

Отмеченные трудности можно легко обойти, если использовать другую форму записи операций. Она называется постфиксной и отличается тем, что знак операции ставится непосредственно за операндами. Такая запись обладает двумя ценными свойствами, благодаря которым ее используют как промежуточную форму представления исходной программы при трансляции:

1. Для записи любого выражения не нужны скобки. Так как оператор непосредственно следует за операндами, участвующими в операции, неопределенность в указании операндов отсутствует. Например, выражение (A + B) * C в постфиксной записи имеет вид: A B + C *.

2. К моменту считывания очередного оператора соответствующие операнды уже прочитаны. Поэтому оператор может быть выполнен без чтения каких-либо дополнительны данных.

Сказанное выше относится к бинарным операциям, однако не трудно распространить результаты рассуждений и на унарные операции. Однако при этом могут возникнуть сложности. Например, знак "–" может стоять в инфиксной записи, указывая как бинарную, так и унарную операцию, и его правильный смысл становится очевидным из контекста. В постфиксной записи сделать это труднее. Унарный минус и другие унарные операции можно представлять двумя способами: либо записывать их как бинарные операции, например, вместо " – B " писать " 0 – B "; либо для обозначения унарных операций вводить новый символ, например, выражение A + (– B + C * E) в постфиксной форме примет вид: A B @ C E * + + .

1.4. ГЕНЕРАЦИЯ КОДА

Возможны три формы объектного кода: абсолютные команды, помещенные в фиксированные ячейки (после окончания компиляции такая программа немедленно выполняется); программа на автокоде (ее потом придется транслировать); программа на языке машины, записанная на внешнюю память (для выполнения она должна быть объединена с другими подпрограммами и затем загружена).

Первый вариант наиболее экономичен в отношении расходуемого времени. Главный недостаток этого варианта состоит в том, что нельзя предварительно и независимо протранслировать несколько подпрограмм и затем объединить их вместе дня выполнения, все подпрограммы должны транслироваться одновременно. Выигрыш во времени оборачивается проигрышем в гибкости.

Проще всего получить объектную программу на автокоде. В этом случае не приходится формировать команды как последовательности битов; можно порождать команды, содержащие символические имена. Более того, можно формировать макроопределение. Это позволяет также уменьшить объем компилятора.

Несмотря на очевидные достоинства, трансляция на автокод обычно считается наихудшим из вариантов. И в самом деле, к процессу трансляции добавляется еще один шаг, который часто требует столько же времени, сколько длится собственно компиляция.

Большинство промышленных компиляторов вырабатывают объектную программу в виде объектного модуля. Как правило, объектный модуль содержит символические имена других программ (подпрограмм), к которым он обращается, и имена своих входных точек, к которым можно обращаться из других программ. Эта объектная программа "объединяется" с теми другими объектными программами, а затем загружается в некоторую область памяти для выполнения.

В этом варианте обеспечивается гораздо большая гибкость, и поэтому во многих системах он и принят в качестве стандартной процедуры. Следует, однако, заметить, что на объединение и загрузку также расходуется время.

Теперь покажем, как генерируются команды для последовательности четверок и постфиксной записи, используя в качестве примера выражение

A * ((A * B + C) – C * B).

Будем считать переменные A, B, C, B целыми.

Генерация кода для последовательности четверок. Для рассматриваемого примера последовательность четверок имеет вид:

*	A	B	I1
+	I1	C	I2
*	C	D	I3
–	I2	I3	I4
*	a	I4	I5

В основе процедуры генерации кода лежит оператор case:

```
procedure ГК; case код операции четверки of
        * : подпрограмма, соответствующая операции *;
        + : подпрограмма, соответствующая операции + ;
        –: подпрограмма, соответствующая операции – ;
        end;
        end;
```

Генерация кода для постфиксной записи. Для рассматриваемого примера постфиксная запись имеет вид: A A B * C + C D * – A *. Операторы и операнды просматриваются последовательно слева направо. Всякий раз, когда просматривается операнд, в стек заносится его имя. А когда встречается операция, генерируются команды для ее выполнения. При этом в качестве описаний операндов используются два верхних описания в стеке; затем эти два описания заменяются описанием результата. При этом необходимо сформировать временное имя T_i, которое заносится в стек.

На практике стек можно отобразить на одномерный массив $S(1)$, $S(2)$, …, $S(n)$. Для указания вершины стека можно использовать индекс i. При записи в стек указатель вершины будет сдвигаться в сторону конца массива, при чтении из стека указатель вершины будет перемещатьсяя в сторону начала массива (рис. 11).

Рис. 11

Для обработки доступен только элемент $S(i)$, т.е. вершина стека. Значение $i = 0$ перед чтением из стека служит признаком того, что стек пуст, а значение $i = n$ перед записью в стек признаком того, что стек переполнен.

2. ОРГАНИЗАЦИЯ ТАБЛИЦ СИМВОЛОВ

Проверка правильности семантики и генерация кода требуют знания характеристик идентификаторов, используемых в программе на исходном языке. Эти характеристики выясняются из описаний и из того, как идентификаторы используются в программе. Вся информация накапливается в таблицах символов.

Таблицы всех типов имеют общий вид (табл. 6). В нашем случае аргументами таблицы являются символы или идентификаторы, а значениями – их характеристики. Когда компилятор начинает трансляцию исходной программы, таблица символов пуста или содержит только несколько элементов для служебных слов и стандартных функций.

В процессе компиляции для каждого нового идентификатора элемент добавляется только один раз, но поиск ведется всякий раз, когда встречается этот идентификатор. Так как на этот процесс уходит много времени, важно выбрать такую организацию таблиц, которая допускала бы эффективный поиск.

Таблица 6

№	Аргумент	Значение
1		
2		
…	…	…
…	…	…
…	…	…
N		

2.1. УПОРЯДОЧЕННЫЕ И НЕУПОРЯДОЧЕННЫЕ ТАБЛИЦЫ

Простейший способ организации таблицы состоит в том, чтобы добавлять элементы в порядке их поступления без каких-либо попыток упорядочения. Поиск в этом случае требует сравнения с каждым элементом таблицы, пока не будет найден подходящий. Для таблицы, содержащей N элементов, в среднем будет выполнено $N/2$ сравнений. Если N велико, такой способ неэффективен. Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены согласно некоторому естественному порядку аргументов. В нашем случае, когда аргументами являются строки символов, наиболее естественным является упорядочение по алфавиту. Эффективным методом поиска в упорядоченном списке является так называемый бинарный поиск. Символ S, который следует найти, сравнивается с аргументом $(n + 1)/2$ в середине таблицы. Если этот элемент не является требуемым, мы должны просмотреть только блок элементов от $(n + 1)/2 + 1$ до n или от 1 до $(n + 1)/2 – 1$ в зависимости от того, больше элемент S или меньше элемента, с которым его сравнивали. Затем мы повторяем процесс над блоком меньшего размера.

Так как на каждом шаге число элементов, которые могут содержать S, сокращается наполовину, максимальное число сравнений равно $\log_2(n + 1)$. Если $n = 2$, нам потребуется самое большее 2 сравнения; если $n = 4$ – то 3; если $n = 8$ – то 4. Для $n = 128$ бинарный поиск требует самое большее 8 сравнений. В то время, как в неупорядоченной таблице в среднем потребуется 64 сравнения.

Сортировка таблицы производится методом упорядоченных вставок.

2.2. ХЕШ-АДРЕСАЦИЯ

Наиболее эффективный и широко применяемый в компиляторах метод при работе с таблицами символов – хеш-адресация. Механизм расстановки состоит из таблицы и хеш-функции (табл. 7). Таб-

лица состоит из N элементов, где N заранее фиксировано. Метод хеш-адресации заключается в преобразовании символа в индекс элемента в таблице. Индекс получается хешированием символа, т.е. выполнением над ним некоторых операций. Если в процессе компиляции встретился объект a, то для поиска его в таблице можно воспользоваться следующим алгоритмом: если объект уже встречался ранее, то $h(a)$ – ячейка в таблице, в которой хранится a. Если объект a ранее не встречался, то $h(a)$ – пустая ячейка, в которую заносится информация для a.

Таблица 7

№	Аргумент	Значение
0		
1		
…	…	…
$h(a)$		
…	…	…
$N-1$		

Возникает, однако, затруднение, если результаты хеширования двух разных символов совпадают. Это называется коллизией. Очевидно, в данной позиции таблицы может быть помещен только один из этих символов, так что мы должны найти свободное место для второго. Желательно иметь такую хеш-функцию, которая распределяла бы объекты равномерно по всей таблице, так чтобы коллизии не возникали слишком часто. Но избежать их совсем практически не удается, поэтому разработчику компилятора следует предусмотреть способы решения задачи коллизии. Существует два таких способа – рехеширование и метод цепочек.

Для простоты предположим, что каждый табличный элемент занимает одно слово. Тогда для таблицы с n элементами значениями функции хеширования h являются целые числа 0, 1, 2, …, $n-1$. Если табличный элемент состоит из k слов, то для вычисления его адреса, необходимо к базовому адресу таблицы добавить произведение значения хеш-функции h на k.

2.3. РЕХЕШИРОВАНИЕ

Предположим, что мы хешируем символ S и обнаруживаем, что другой символ уже занял элемент с номером h. Возникает коллизия. Тогда сравниваем S с элементом в ячейке $h + p_1$ (для некоторого целого p_1). Если снова возникает коллизия, сравниваем S с элементом $h + p_2$. Это продолжается до тех пор, пока не встретится ячейка $h + p_i$, которая либо пуста, либо содержит S, либо совпадает с ячейкой h ($p_i = 0$). В последнем случае считаем, что таблица переполнена.

Таким образом, если возникло i коллизий, будет выполнено $i + 1$ сравнение с элементами таблицы. Элементы p_i должны выбираться так, чтобы ожидаемое число сравнений было невелико, но при этом рассматривалось большее число элементов. В идеальном случае p_i должны охватывать целые числа 0, 1, …, $n-1$.

Существуют несколько способов рехеширования. Рассмотрим некоторые из них.

2.3.1. Линейное рехеширование

Это простейший метод рехеширования, состоящий в том, чтобы принять $p_1 = 1$, $p_2 = 2$, $p_3 = 3$ и так далее. В этом случае мы двигаемся по таблице вперед относительно значения $h(a)$ до тех пор, пока не исчезнет коллизия. Если достигнута позиция $n-1$, переходим на позицию 0. Метод прост для выполнения, однако, если уже возник конфликт, то занятые позиции имеют тенденцию скапливаться. Метод выражается формулой

$$h_i(a) = (h(a) + i) \bmod n, \quad 1 \le i \le n-1$$

2.3.2. Случайное рехеширование

Этот метод снимает проблему скопления, которая свойственна линейному рехешированию, за счет выбора в качестве p_i достаточно использовать самый элементарный генератор случайных чисел, выдающий все числа в интервале от 0 до $n - 1$ по одному разу. Каждый раз, когда используется генератор случайных чисел, он устанавливается в одно и то же состояние. Таким образом, когда происходит обращение к h, генерируется одна и та же последовательность $p_1, p_2, \ldots$.

2.3.3. Рехеширование сложением

В этом случае принимаем $p_i = i * h$, где h – исходный хеш-индекс. Таким образом рассматриваются элементы таблицы с номерами h, $2h$, $3h$, $4h$ … . Этот метод хорош, если размерность таблицы n будет простым числом, то все последовательности полностью покроют все возможные индексы 1, …, $n - 1$. Метод описывается формулой:

$$h_i(a) = (ih(a)) \bmod n, \quad 1 \le i \le n - 1 .$$

2.4. МЕТОД ЦЕПОЧЕК

Метод цепочек использует хеш-таблицу, элементы которой первоначально равны 0, собственно таблицу символов, вначале пустую, и указатель УК, который указывает на текущее положение последнего элемента в таблице символов. Элементы таблицы символов имеют дополнительное поле CHAIN, которое может содержать 0 или адрес другого элемента таблицы символов. Начальное состояние таблицы приведено на рис. 12.

Хеш-функция, примененная к символу, дает индекс указателя в хеш-таблице. Указатель либо равен 0, либо указывает на первый элемент таблицы символов с данным значением хеш-функции. Поле CHAIN каждого элемента используется для того, чтобы связать в цепочку элементы, для которых хеширование символа приводит к тому же самому указателю. Например, в таблицу необходимо записать символ $S1$. Функция хеширования вырабатывает адрес элемента хеш-таблицы, например 4. Содержимое этой ячейки равно 0. Тогда выполняется следующее:

1. Прибавляем 1 к УК.
2. Вносим элемент ($S1$, значение, 0) в позицию таблицы символов, на которую указывает УК.
3. Заносим содержимое УК в указатель 4 хеш-таблицы.

№	Хеш-таблица
0	
1	
…	
$N - 1$	

№	Аргумент	Значение	CHAIN
1			
2			
…			
N			

УК = 0

Рис. 12

№	Хеш-таблица
0	
1	2
2	
3	3
4	1
5	
6	4

№	Аргумент	Значение	CHAIN
1	$S1$	…	0
2	$S2$	…	0
3	$S3$	…	0
4	$S4$	…	0

УК = 4

Рис. 13

Пока поступают символы, хеширование которых дает индексы разных указателей, они заносятся в таблицу аналогичным образом. Так, если мы записываем в таблицу символы *S*2, *S*3, *S*4, хеширование которых дает ссылки на указатели 1, 3, 6, таблица примет вид, представленный на рис. 13.

В конце концов, поступит символ *S*5, который ссылается на указатель, использовавшийся ранее, например на 6. Вот здесь и начинает действовать поле CHAIN. Символ *S*5 записывается в таблицу символов и добавляется к концу цепочки для этого указателя (рис. 14).

№	Хеш-таблица
0	
1	2
2	
3	3
4	1
5	
6	4

№	Аргумент	Значение	CHAIN
1	*S*1	…	0
2	*S*2	…	0
3	*S*3	…	0
4	*S*4	…	5
5	*S*5	…	0

УК = 5

Рис. 14

На рис. 15 приведена блок-схема алгоритма занесения одного элемента *S* в таблицу символов с помощью метода цепочек.

Пусть необходимо внести в таблицу символы *S*6, *S*7, *S*8, которые ссылаются на указатели 4, 3, 3, соответственно (рис. 16).

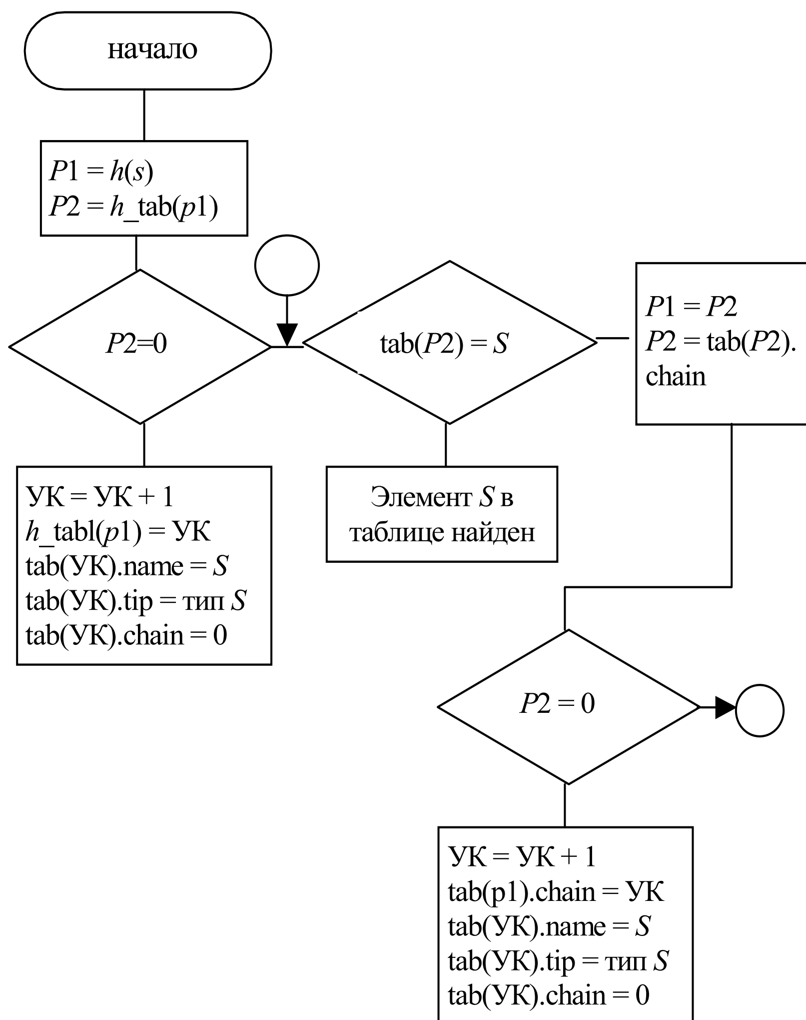


Рис. 15

№	Хеш-таблица
0	
1	2
2	
3	3
4	1
5	
6	4

№	Аргумент	Значение	CHAIN
1	*S*1	…	6
2	*S*2	…	0
3	*S*3	…	7
4	*S*4	…	5
5	*S*5	…	0
6	*S*6	…	0
7	*S*7	…	8
8	*S*8	…	0

УК = 8

Рис. 16

3. ОПТИМИЗАЦИЯ КОДА

Рассмотрим некоторые методы машинно-независимой оптимизации кода. Мы не будем стремиться к детальному описанию какого-либо из этих методов. Вместо этого мы дадим словесное описание и проиллюстрируем основные понятия примерами.

Одним из важных источников оптимизации кода является удаление *общих подвыражений*, которые встречаются в нескольких местах программы и вычисляют одно и тоже выражение. Рассмотрим, например предложение:

```
VAR x,y: ARRAY [1..10,1..10]  OF  INTEGER;
                                  …
FOR i : = 1 TO 10 DO
x [ i , 2*j–1 ] : = y [ i , 2*j ];

                                  …
```

Выражение 2*j является общим подвыражением. Оптимизирующий компилятор должен только один раз сгенерировать код, вычисляющий это умножение, и использовать его результат в обоих местах.

Общие подвыражения обычно обнаруживаются при анализе промежуточной формы представления программы (рис. 17). Следует отметить, что в первоначальном варианте требуется выполнить 161 операцию.

Если мы исследуем эту последовательность четверок, то обнаружим, что четверки 5 и 11 совпадают, за исключением имени получаемого промежуточного результата. Обратите внимание, что операнд j не меняет своего значения между четверками 5 и 11. Невозможно достичь четверки 11, не проходя предварительно четверку 5, поскольку они расположены на одном линейном участке.

1)	: =	#1		i	инициализация цикла
2)	>	i	#10	(19)	
3)	–	i	#1	i1	вычисление индексов для x
4)	*	i1	#10	i2	
5)	*	#2	j	i3	
6)	–	i3	#1	i4	
7)	–	i4	#1	i5	
8)	+	i2	i5	i6	
9)	–	i	#1	i8	вычисление индексов для y
10)	*	i8	#10	i9	
11)	*	#2	j	i10	
12)	–	i10	#1	i11	
13)	+	i9	i11	i12	
14)	: =	y[i12]		x[i6]	операция присваивания

15)	+	#1	i	i14	конец цикла
16)	: =	i14		i	
17)	GOTO			(2)	
18)	…				следующая операция

Рис. 17

Таким образом, четверки 5 и 11 вычисляют одно и то же значение. Это означает, что мы можем удалить четверку 11 и заменить любые обращения к ее результату (i9) на обращение к переменной i3, которая является результатом четверки 5. Эта модификация позволяет избежать дублирования вычислений подвыражения 2*j, которое мы выделили как общее подвыражение при анализе исходной программы.

После замены i3 на i10 мы обнаружим, что четверки 6 и 12 также совпадают, за исключением имени результата. Следовательно, мы можем удалить четверку 12 и заменить переменную i11 всюду, где она используется, на переменную i4. Аналогично четверки 10 и 11 также могут быть удалены, поскольку они эквивалентны четверкам 3 и 4. В результате получим новую последовательность четверок (рис. 18), которая предполагает выполнение всего 121 операции.

Обратите внимание, что общее количество четверок сокращено с 17 до 13. Поскольку операции во всех используемых здесь четверках займут, вероятно, примерно одинаковое время на обычном компьютере, то мы также сократим общее время выполнения программы.

Другим источником оптимизации кода является удаление *инвариантов цикла*. Так называются подвыражения внутри цикла, результирующие значения которых не изменяются внутри цикла при переходе от одной итерации к другой. Таким образом, эти значения могут быть вычислены только один раз перед входом в тело цикла вместо того, чтобы вычислять их заново перед каждой итерацией. Поскольку для большинства программ основное время работы приходится на выполнение циклов, экономия времени от подобной оптимизации может быть весьма существенной.

1)	: =	#1		i	инициализация цикла
2)	>	i	#10	(14)	
3)	–	i	#1	i1	вычисление индексов для x
4)	*	i1	#10	i2	
5)	*	#2	j	i3	
6)	–	i3	#1	i4	
7)	–	i4	#1	i5	
8)	+	i2	i5	i6	
9)	+	i2	i4	i12	вычисление индексов для y
10)	: =	y[i12]		x[i6]	операция присваивания
11)	+	#1	i	i14	конец цикла
12)	: =	i14		i	
13)	GOTO			(2)	
14)	…				следующая операция

Рис. 18

Примером инварианта цикла является вычисление выражения 2*j. Результат вычисления этого выражения зависит только от операнда j, значение которого не изменяется во время выполнения цикла. Таким образом, мы можем поместить четверку 5 непосредственно перед началом выполнения цикла. Аналогичные соображения – относительно четверок 6 и 7.

1)	*	#2	j	i3	вычисление инвариантов
2)	–	i3	#1	i4	
3)	–	i4	#1	i5	
4)	: =	#1		i	инициализация цикла
5)	>	i	#10	(14)	
6)	–	i	#1	i1	вычисление индексов для x
7)	*	i1	#10	i2	
8)	+	i2	i5	i6	
9)	+	i2	i4	i12	вычисление индексов для y
10)	: =	y[i12]		x[i6]	операция присваивания
11)	+	#1	i	i14	конец цикла
12)	: =	i14		i	
13)	GOTO			(5)	
14)	…				следующая операция

Рис. 19

Примечание: необходимо выполнить 94 операции.

В результате получим новую последовательность четверок (рис. 19). Общее количество четверок остается тем же, но количество четверок в цикле уменьшилось с 12 до 9. Каждое выполнение предложения FOR вызывает десятикратное повторение тела цикла. Это означает, что общее количество операций, необходимых для выполнения FOR, сократилось со 121 до 94.

Общее количество операций, приходящееся на одно выполнение предложения FOR, по сравнению с начальным вариантом, сократилось со 161 до 94, что существенно уменьшило выполнение программы.

Существуют также и более тонкие методы обработки общих подвыражений и инвариантов цикла, чем описанные выше. Можно ожидать, что благодаря этим методам может быть получен еще более оптимизированный вариант кода.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Какова роль лексического анализатора?
2. Что такое лексемы?
3. Для чего нужна кодировочная таблица?
4. Для чего служит таблица символов?
5. Как организуется таблица символов?
6. В чем состоит метод бинарного поиска?
7. Что такое хеширование?
8. Как работает метод цепочек?
9. Какие известны способы рехеширования?
10. Чем отличаются способы нисходящего и восходящего грамматического разбора?
11. Какие известны методы синтаксического анализа?
12. В чем сущность метода рекурсивного спуска?
13. Для чего применяется измененный способ записи БНФ?
14. В каком виде представляется программа на выходе синтаксического анализатора?
15. В чем сущность метода операторного предшествования?
16. Как сравниваются известные способы организации таблиц символов?
17. В чем суть методов бинарного поиска и упорядоченных вставок?
18. Каковы достоинства хеш-адресации, и каковы ее недостатки?
19. Какие известны способы рехеширования?
20. В чем преимущество метода цепочек по сравнению с рехешированием?
21. Какая информация хранится в таблице символов?
22. Как записать информацию о переменных?
23. Разработать грамматику языка и алгоритмы отдельных этапов трансляции для фрагментов программ, приведенным в табл. 8.

Таблица 8

```
1. ACCEPT *; A
   if  (A.LE.0)  S=S+A
   S=A*100–K
```

```
2. MAX=0
   if (X.GT.MAX) MAX=X
   TYPE *,  MAX
```

```
3. D=B*B=–4*A*C
   if (D.EQ.O) T=–B/(2*A)
   TYPE*, T
```

```
4. S1:=A MOD B;
   S2:=A–((A DIV B)*B);
   if S1=S2 then write ('BCE')
```

```
5. if x>0 then y:=l
  else if x<0 then y:=2
  else y:=0
```

```
6. if Y>MIN & Y<MAX then
   PUT LIST (Y);
   else X=X+1;
```

```
7. Z=Z+1;
   if Z>N then DO;
   Z=Z/N;
   PUT DATA (Z);
   end;
```

```
8. X=X0+(i–l)*H
  if (X.LT.O)  Y=X*X
  if  (X.GT.O)  Y=2*X
```

```
9. if i MOD 2=0 THEN Z:=Z*X;
   i:=i DIY 2;
   X:=X*X*X
```

```
10. READ(B);
    if B>L THEN begin
   A:=A–Z;
   A:=A+B end
   else B:=L+B
```

```
11. if (X+Y)<>0 THEN
   A:=(X*X+Z*Z)/(1 +1/(X–Y*Z))
   else A:=0;
```

```
12. A:=-3*C;
   B:=Y*Y*X;
   if (A<0) AND (B>0) THEN
```

```
writeln (A)
```

```
C:=–A+B
else C:=A+B
```

```
13. if  T>EPS THEN begin
    K:=K+2;
      T:=–T*SX/(K*(K–1));
    S:=S+T
    end
```

```
14. if N>0 THEN
   M:=–M+l/N;
   B:=M*N+3;
    if B=0 THEN write (N)
```

```
15. if  (2.5+0.68<=2.8) OR Y
   AND X OR Z AND NOT Y
   THEN
   writeln ('верно')
```

```
16. READ(N);
    M:=0;
    If f=1 THEN writeln (M)
    else M:=M+1/N
```

```
17. RESET (F,’F.DAT’);
    SUM:=SUM+F^;
    GET (F)
```

```
18. REWRITE (F, 'F.DAT');
    F^:=K*K;
    PUT (F)
```

```
19. READ (x, y);
   A:=x/(y*y*x*x/(y+x/3))
```

```
20. READ (H, B, M);
    PI:=3.14;
    V:=PI*H*
    *(B*B+M*M+B*M)/3
```

```
21. A=Z*Z;
    B=l+A/(3*Z+A/5);
    if B>5 THEN writeln (B)
```

```
22. if x<0.5 THEN y=2*x*x–x
    else y=x*x/(x–0.1);
    write (y)
```

```
23. READ (A, B);
    IF A>B THEN
    X=2*A+2/B+4;
    IF A<=B THEN
    x=(A+B)*(A–B);
    writeln (x)
```

```
24. READ (I);
    if (I>4) OR (I<0) THEN
    writeln ('ошибка')
```

```
25. if A<0 THEN A=–A;
    if B<0 THEN B=–B;
    SA:=(A+B)/2;
    SB:=A*B/2
```

```
26. READLN (Y);
    if Y<0 THEN
    Z:=Y–3*Y*Y/(Y+1)
    else if Y=0 THEN Z:=0
    else Z:=100*Y
```

КУРСОВАЯ РАБОТА
ПО ДИСЦИПЛИНЕ "ЛИНГВИСТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ САПР"

Основная цель – закрепление навыков создания лингвистического обеспечения САПР, описание языка с помощью формальной грамматики, разработка алгоритма и реализация простых вариантов трансляторов, изучение различных методов синтаксического анализа.

Общие требования к оформлению курсовых работ по дисциплине "Лингвистическое и программное обеспечение САПР" для студентов 2 курса специальности 230104.

В отчете по курсовой работе необходимо отразить следующие разделы:

1. Пояснительная записка, включающая:

1.1. Задание на проектирование (в том числе дата принятия задания к исполнению, подпись студента).

1.2. Содержание (название разделов, подразделов с указанием страниц; титульный лист не нумеруется, но считается).

1.3. Введение (отразить общее назначение программы, указать дату разработки, кем разработана программа, ее название).

1.4. Описание процесса решения задачи.

1.5. Блок-схема основной программы и процедур.

2. Распечатка программных модулей.

3. Описание программы, включающее:

3.1. Назначение и общее описание программы.

3.2. Описание логической структуры программы.

3.3. Способ обращения к программе (дать краткую характеристику операционной среды: как обратиться к программе, как получить загрузочный модуль, как запустить программу на выполнение).

3.4. Перечень технических средств.

4. Описание входных и выходных данных.

5. Текстовые примеры работы программы (контрольные примеры при верных исходных данных и ошибочных).

Методические указания

Взаимодействие между компонентами компилятора может осуществляться разными способами. На рис. 20, *а* показано, что лексический анализатор (ЛА) считывает исходную программу (ИП) и представляет ее в виде файла лексем. Синтаксический анализатор (СА) читает этот файл и выдает внутреннее представление программы (ВП) программы. Наконец, этот файл считывается генератором кода (ГК), который создает объектный код программы. Компилятор такого вида называется трехпроходным, так как программа считывается трижды (исходная программа, лексемы, внутреннее представление).

На рис. 20, *б* изображена структура однопроходного компилятора. В этом случае синтаксический анализатор выступает в роли управляющей программы, вызывая лексический анализатор и генератор кода, организованные в виде процедур. Синтаксический анализатор постоянно обращается к лексическому анализатору, получая от него лексему за лексемой из просматриваемой программы до тех пор, пока не построит новый элемент внутреннего представления, после чего обращается к генератору кода, который создает объектный код для этого фрагмента программы.

Каждый из этих способов организации компиляторов имеет свои преимущества. В трехпроходном компиляторе достигается высокая гибкость за счет независимости каждой фазы трансляции. С другой стороны, если требуется достичь высокой скорости транслирования, используют однопроходный компилятор, в котором исходная программа считывается один раз.

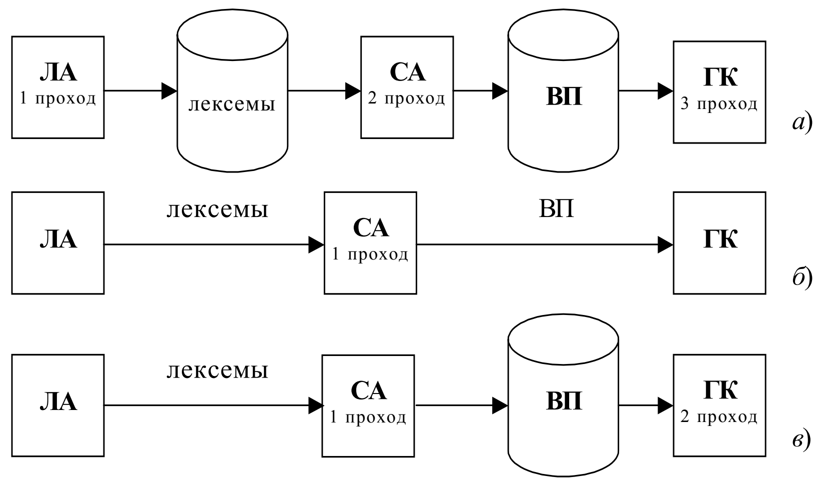


Рис. 20

Промежуточное положение между описанными двумя вариантами занимает двухпроходный компилятор (рис. 20, *в*). В этом случае синтаксический анализатор, вызывая лексический анализатор, получает лексемы и строит файл во внутреннем представлении. Генератор кода считывает этот файл и создает объектный код.

Варианты заданий

1. *Разработать однопроходный транслятор с исходного языка на язык Фортран*:

```
программа X;
переменные y,z: вещественные;
   i: целые;
z=0;
цикл i от 1 до 10 выполнить
z=z*y;
вывод z;
конец.
```

2. *Разработать двухпроходный транслятор с исходного языка на язык Фортран*:

```
program EF:
var i,n: integer;
h: real;
begin
read(n);
h:=0;
for i:=n downto 1 do h:=h+1/i;
writeln(h)
              end.
```

3. *Разработать трехпроходный транслятор с исходного языка на язык ПЛ-1*:

```
program g;
var u, v, max. min: real;
begin
read(u,v);
max:=10;
min:=0;
if u>v then
```

```
begin
if u>max then max:=u;
if v<min then min:=v
end
     else
begin
  if v>max then .max:=v;
    if u<min then min:=u
  end;
write(max,min)
end.
```

4. *Разработать однопроходный транслятор с исходного языка на язык Паскаль*:

```
программа B;
переменные c, a, k: вещественные;
                  i: целые;
s=0;
i=1;
ввод a;
начало цикла
     s=s+1/i;
i=i+1;
      если s>a то закончить цикл;
вывод i,s
конец.
```

5. *Разработать двухпроходный транслятор с исходного языка на язык Паскаль*:

```
программа Z;
переменные a,b,c: вещественные;
начало
  ввод a,b,c;
  если a>b и b>c то выполнить
(a=2*a;b=2*b;c=2*c)
иначе выполнить ( a=|a|; b=|b|; c=|c| );
вывод a,b,c;
конец.
```

6. *Разработать трехпроходный транслятор с исходного языка на язык Паскаль*:

```
программа. W;
переменные x: вещественные;
i: целые;
ввод i,x;
 если i равен
    [ 0 то x=0,
      1 то x=sin(x),
      2 то x=cos(x),
      3 то x=x*x]
x=x/2;
      вывод x;
      конец.
```

7. *Разработать однопроходный транслятор с исходного языка на язык ассемблера*:

```
программа С;
переменные x,y,z:целые;
ввод x,y:
если x>y то z=x–y
                иначе z=y–x+l;
              вывод z;
конец.
```

8. *Разработать двухпроходный транслятор с исходного языка на язык ассемблера*:

```
программа М;
переменные i,j: целые;
ввод i,j;
если i>j то вывод I иначе (i=i+j, вывод i);
конец.
```

9. *Разработать трехпроходный транслятор с исходного языка на язык ассемблера*:

```
program К;
    label 1;
 var f, i, n: integer;
    begin
       read(n);
 f:=1;    i:=1;
1: f:=f*i;
    i:=i+1;
  if i<=n then goto 1;
writeln(f)
end.
```

10. *Разработать однопроходный транслятор с исходного языка на язык Бейсик*:

```
программа К;
переменные n, i, f: целые;
   f=l;
      ввод (n);
      цикл i=1,n выполнить f=f*i;
вывод (f)
конец.
```

11. *Разработать двухпроходный транслятор с исходного языка на язык Бейсик:*

```
real p,x
integer i,n
accept *,x,n
p=1
do 15 I=1,n
p=p*(1–x/i)**2
   15  continue
stop
end
```

12. *Разработать трехпроходный транслятор с исходного языка на язык Бейсик*:

```
real x,y
accept *,x,n
if (x.LT.0) y=x**2
```

```
if (x.GE.0) y=x**3
type *,y
stop
end
```

13. *Разработать трехпроходный транслятор с исходного языка на язык Фортран*:

```
программа P;
переменные s: вещественные;
              i, n: целые;
s=1;
ввод n;
              цикл 1 от 1 до n с шагом 2
выполнить s=s*i/(i+1);
вывод s;
конец.
```

14. *Разработать двухпроходный транслятор с исходного языка на язык Паскаль*:

```
программа E;
переменные h: вещественные;
              n: целые;
h:=0;
читать(n);
пока n > 0 выполнить
(h:=h+l/n; n:=n–l);
печатать (h)
конец.
```

15. *Разработать однопроходный транслятор с исходного языка на язык ПЛ-1*:

```
integer m,n
do 10 n=11,49,2
m=n**2
type *,n,m
   10  continue
stop
end
```

16. *Разработать однопроходный транслятор с исходного языка на язык Фортран*:

```
program K16;
var i,n: integer;
    p,a:.rea1;
begin
  read(n);
  p:=1;
for i:=l to n do p:=p*(a+i–1);
write(p)
end.
```

17. *Разработать двухпроходный транслятор с исходного языка на язык ПЛ-1*:

```
программа PC;
переменные x, y: вещественные;
ввод (x,y);
   если x>y то [ y=(x+y)/2;x=x*y/2]
```

```
иначе [ x=(x+y)/2; y=x*y/2 ];
вывод (x, y)
конец.
```

18. *Разработать однопроходный транслятор с исходного языка на язык Паскаль*:

```
real x,s
integer k,i,n
  accept *,x,n
   s=0
   do 10 i=1,n
   k=2*i+1
   s=s+cos(k*x)/k
10                continue
        type *,s
        stop
        end
```

19. *Разработать трехпроходный транслятор с исходного языка на язык Си*:

```
процедура X;
переменные s1,s2,a, b: вещественные;
              i,j:целые:
s1:=0;
s2:=0;
ввод (a,b);
   цикл i от 0 до 10 шаг 2 выполнить
     [
       s1:=s2+a;
       s2:=0;
       цикл j от 1 до 20 шаг l выполнить
       s2:=s2+b;
 ]
    передача (sl)
конец.
```

20. *Разработать двухпроходный транслятор с исходного языка в язык Си*:

```
procedure K;
var i,n,f: integer;
begin
read(n);
f:=1;
i:=1;
while i<=n do
  begin
     f:=f*i;
     i:=i+1
  end;
write(f)
             end.
```

21. *Разработать трехпроходный транслятор с исходного языка на язык Паскаль*:

```
main( )
 {  int i;
```

```
 double h,b,a,m,n,d,x;
 scanf ("%le%le",&a,&b);
h=(b-a)/10;
for(I=0; i<=10;i++)
{   x=a+h*i;
m=(2*h*h-4)/(2*h);
n=(2–h)/(2+h);
d=m*h*h–m*n;
printf ("\n%le,%le",x,d);
}
}
```

22. *Разработать однопроходный транслятор с исходного языка на язык Бейсик*:

```
программа Z;
переменные y,k: вещественные;
      i,n: целые;
ввод (k);
ввод (n);
    цикл i от 1 до n выполнить
k:=k*y;
                         вывод (k)
                       конец.
```

23. *Разработать однопроходный транслятор с исходного языка на язык Си*:

```
программа AB;
переменные i, n: integer;
                      h: real;
начало
   читать(n);
    h=0;
    цикл i от n до 1 с шагом –1
        выполнять
           h=h+1/i;
     вывести(h)
 конец.
```

24. *Разработать двухпроходный транслятор с исходного языка на язык Си*:

```
программа A;
переменные k:целые;
                      t, eps, sx, s: вещественные;
начало
   читать(eps,sx);
    s=0;    k=1;     t=1;
    пока |t|>eps выполнять
           { k:=k+2;
              t:=–t*sx/(k*(k–1));
              s:=s+t
            };
     вывести(h)
 конец.
```

25. *Разработать трехпроходный транслятор с исходного языка на язык Си*:

```
программа z;
```

```
переменные i:целые;
                x: вещественные;
начало
  читать(i);  читать(x);
  пока (i<4) и (i>0) выполнять
        {
          если i
               =0 то x:=0;
               =1 то x:=sin(x);
               =2 то x:=exp(x);
               =3 то x:=cos(x);
               =4 то x:=ln(x)
             все;
   вывести(x);
  }
 конец.
```

26. *Разработать однопроходный транслятор с исходного языка на язык Паскаль*:

```
integer i,j
real c,p
do 2 i=1,10
do 2 j=1,10
c=a+i*j
if (c.ge.1.and.c.le.10) p=p*c
  2    continue
stop
end
```

27. *Разработать двухпроходный транслятор с исходного языка на язык Паскаль*:

```
integer f,i,n
accept *,n
f=1
i=1
do 1 i=1,n,1
f=f*i
1         continue
type *,f
stop
end
```

28. *Разработать трехпроходный транслятор с исходного языка на язык Паскаль*:

```
программа xy;
переменные k,l:целые;
                sx,sy,x, y: вещественные;
начало
   sx=0;
   sy=0;
   цикл k от 0 до 5 шаг 1 выполнять
        {
          sx:=sy+x;
          sy:=0;
```

```
    цикл l от 1 до 10 шаг 1 выполнять
            sy:=sy+y
          };
    вывести(sx,sy)
 конец.
```

29. *Разработать однопроходный транслятор с исходного языка на язык Си*:

```
программа l;
переменные i,n,f:целые;
начало
  читать(n);
    f=1;      i=1;
   пока i<=n выполнять
         { f:=–f*i;
            i:=i+1
          };
    вывести(f)
                       конец.
```

30. *Разработать двухпроходный транслятор с исходного языка на язык ПЛ-1*:

```
программа с;
переменные i:целые;
                  a, b, h, m, n, d, x: вещественные;
начало
  читать(a,b);
   h=(b–a)/10;
   цикл (i=0; i<=10; i++)
       {    x:=a+h*i;
            m:=(2*h*h–4)/(2*h);
            n:=(2–h)/(2+h);
            d:=m*h*h–m*n;
            вывести(h)
          };
                        конец.
```

31. *Разработать трехпроходный транслятор с исходного языка на язык ПЛ-1*:

```
main()
 {
int i,j;
double a,b;
scanf("%le%le",&a,&b);
s1=0;
s2=0;
for (i=0; i<=10; i=i+2)
  {
    s1=s2+a;
    s2=0;
    for (j=0; j<=20; i++)
         s2=s2+b;
   }
  printf ("\n%le%le",s1,s2)
}
```

32. *Разработать однопроходный транслятор с исходного языка на язык Си*:

```
программа g;
переменные i: integer;
                    u, v, min, max: real;
начало
   читать(u,v);    min:=0;    max:=10;
    цикл i от 1 до 10 с шагом 1 выполнять
      {  если u>v то
           {
             если u>max то max:=u;
             если v<min то min:=v; }
          иначе
           {
             если v>max то max:=v;
             если u<min то min:=u;
           }
   вывести(max,min);
  }
конец.
```

33. *Разработать двухпроходный транслятор с исходного языка на язык Си:*

```
real h
integer i,n
accept *,n
h=1
do 2 i=n,1,–1
h=h+1/i
  2     continue
type *,f
stop
end
```

34. *Разработать трехпроходный транслятор с исходного языка на язык Си*:

```
real s
integer i,n
accept *,n
s=1
do 3 i=1,n,2
s=s*i/(i+1)
  3     continue
          type *,s
          stop
          end
```

35. *Разработать однопроходный транслятор с исходного языка на язык Си*:

```
программа xz;
переменные k, i, n:целые;
                    s, x: вещественные;
начало
   читать(x,n);
    s=0;
    цикл i=1 до n шаг 1 выполннять
```

```
  {  k:=2*i+1;
     s:=s+cos(k*x)/k;
  }
     вывести(s);
                конец.
```

Примечание: приведенный в задании пример программы определяет структуру программы на исходном языке; тип и структуру операторов исходного языка.

Количество идентификаторов, операторов, порядок их следования может быть любым.

Разрабатываемый транслятор должен распознавать орфографические (неописанный идентификатор и повторное описание идентификатора) и синтаксические ошибки, а также выдавать сообщения о них.

Варианты заданий отличаются используемым методом синтаксического анализа.

ЗАКЛЮЧЕНИЕ

В рамках дисциплины "Лингвистическое и программное обеспечение САПР" студенты выполняют лабораторные работы, посвященные отдельным этапам трансляции. Все необходимые материалы представлены в данном учебном пособии. Рассмотрены основные подходы к созданию транслирующих процедур. Показаны основные способы организации трансляторов: однопроходных, двухпроходных, трехпроходных. Приведены основные методы организации таблиц символов и их взаимодействие с основными частями транслятора: механическим анализатором, синтаксическим анализатором и генератором кода. Рассмотрены основные методы машинно-независимой оптимизации кода и на их примере – организация внутреннего представления программы, в том числе циклических процедур. Обобщая полученные знания, студент выполняет курсовую работу.

В учебном пособии приведены алгоритмы различных методов синтаксического анализа и организации таблиц символов. Это поможет студентам в разработке отдельных процедур транслирующих программ.

СПИСОК ЛИТЕРАТУРЫ

1. Ахо, А., Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М. : Издательский дом «Вильямс», 2001.
2. Ахо, А. Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. – М. : Мир, 1976. – Т. 1, 2.
3. Бек, Л. Введение в системное программирование / Л. Бек. – М. : Мир, 1988.
4. Грис, Д. Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – М. : Мир, 1975.
5. Маккиман, У. Генератор компиляторов / У. Маккиман, Дж. Хорнинг, Д. Уортман. – М. : Статистика, 1980.
6. Рейуорд-Смит, В.Дж. Теория формальных языков. Вводный курс / В.Дж. Рейуорд-Смит. – М. : Радио и связь, 1988.
7. Разработка компиляторов : лабораторные работы / сост. И.Л. Коробова. – Тамбов : Тамб. гос. техн. ун-т, 1997. – 28 с.
8. Проектирование трансляторов : методические указания / сост. : И.Л. Коробова, Д.В. Абрамов. Тамбов :
Изд-во Тамб. гос. техн. ун-та, 2000. – 28 с.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ …………………………………………….. 3
1. ТЕОРИЯ ТРАНСЛЯЦИИ ……………………..…. 5
1.1. Лексический анализ ……………………….. 5
1.2. Синтаксический анализ ……………………. 10
1.2.1. Метод рекурсивного спуска ………… 12
1.2.2. Метод операторного предшествования 23
1.3. Внутреннее представление программы ……. 33
1.3.1. Последовательность четверок ………... 34
1.3.2. Постфиксная запись ………………… 35
1.4. Генерация кода ……………………………. 37
2. ОРГАНИЗАЦИЯ ТАБЛИЦ СИМВОЛОВ ……… 40
2.1. Упорядоченные и неупорядоченные таблицы 41
2.2. Хеш-адресация ……………………………… 42
2.3. Рехеширование ……………………………… 43
2.3.1. Линейное рехеширование ……………. 44
2.3.2. Случайное рехеширование …………… 44
2.3.3. Рехеширование сложением …………... 44
2.4. Метод цепочек ……………………………. 45
3. ОПТИМИЗАЦИЯ КОДА …………………….. 49
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ ………… 55
КУРСОВАЯ РАБОТА ПО ДИСЦИПЛИНЕ "ЛИНГВИСТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ САПР" …………………………………………….. 58
ЗАКЛЮЧЕНИЕ ………………………………………. 76
СПИСОК ЛИТЕРАТУРЫ ……………………………… 77

ДЛЯ ЗАМЕТОК