

И. А. КОРОБОВА

ТЕОРИЯ ТРАНСЛЯЦИИ

**Учебное электронное издание
на компакт-диске**

Тамбов
Издательство ФГБОУ ВПО "ТГТУ"
2014

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Тамбовский государственный технический университет"

И. Л. КОРОБОВА

ТЕОРИЯ ТРАНСЛЯЦИИ

Методические указания
для бакалавров 2 курса дневного отделения
специальности 230100 и магистрантов 5 курса направления 230100

Учебное электронное издание
комбинированного распространения



Тамбов
Издательство ФГБОУ ВПО "ТГТУ"
2014

УДК 004.4'4(076)
ББК 3973-018-5-05
К68

Рекомендовано Редакционно-издательским советом университета

Р е ц е н з е н т ы :

Кандидат технических наук,
доцент ФГБОУ ВПО "ТГУ им. Г. Р. Державина"

В. П. Дудаков

Доктор технических наук, профессор ФГБОУ ВПО "ТГТУ"
M. Н. Краснянский

Коробова, И. Л.

К68 Теория трансляции [Электронный ресурс] : метод. указания /
И. Л. Коробова. – Тамбов : Изд-во ФГБОУ ВПО "ТГТУ", 2014. –
64 с. – 1 электрон. опт. диск (CD-ROM). – Системные требова-
ния : ПК не ниже класса Pentium II ; CD-ROM-дисковод 26,0 Mb
RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

Рассмотрены общие сведения по теории формальных грамматик
и языков, приведены основные подходы к построению транслирую-
щих программ.

Предназначены для бакалавров 2 курса дневного отделения спе-
циальности 230100 "Информатика и вычислительная техника" и ма-
гистрантов направления 230100 программы "Модели, методы и про-
граммное обеспечение анализа проектных решений", изучающих
дисциплины "Лингвистические средства вычислительных систем",
"Теория языков программирования и методы трансляции".

УДК 004.4'4(076)
ББК 3973-018-5-05

Все права на размножение и распространение в любой форме остаются за разработчиком.
Нелегальное копирование и использование данного продукта запрещено.

© Федеральное государственное бюджетное
образовательное учреждение высшего
профессионального образования
"Тамбовский государственный технический
университет" (ФГБОУ ВПО "ТГТУ"), 2014

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 создаётся 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.

Приведённые подходы будут полезны для бакалавров 2-го курса специальности 230100 "Информатика и вычислительная техника" при выполнении лабораторных и курсовой работы по дисциплине "Лингвистические средства вычислительных систем" и магистрантов 5-го года обучения специальности 230100 "Информатика и вычислительная техника" при выполнении лабораторных работ по дисциплине «Теория языков программирования и методы трансляции».

1. ТЕОРИЯ ФОРМАЛЬНЫХ ГРАММАТИК И ЯЗЫКОВ

План:

1. Грамматика.
2. Формальные определения грамматики и языка.
3. Классификация грамматик.
4. Синтаксические деревья.

Ключевые слова: синтаксис языка, семантика языка, грамматика, формальный язык, синтаксический язык.

1.1. ГРАММАТИКА

Грамматика языка является формальным описанием его синтаксиса или формы, в которой записаны отдельные предложения программы или вся программа. Грамматика не описывает семантику или значения различных предложений.

В качестве иллюстрации разницы между синтаксисом и семантикой рассмотрим два предложения:

$$i := j + k \text{ и } i := x + y ,$$

где x , y являются действительными переменными, а i , j , k целыми.

Эти два предложения имеют одинаковый синтаксис. Оба являются операторами присваивания. Но с точки зрения семантики, эти предложения разные, так как компилируются в совершенно различные последовательности машинных команд.

В первом случае складываются целые переменные j и k и результат присваивается переменной i .

Во втором случае необходимо сложить вещественные переменные x и y , а результат преобразовать к целому виду и полученное значение присвоить переменной i .

В качестве примера мы будем использовать программу на языке Паскаль, изображённую на рис. 1, однако обсуждаемые концепции и подходы приложимы и к другим языкам. Существует несколько различных форм записи грамматик, среди которых мы рассмотрим форму Бекуса–Наура (БНФ). БНФ не самое мощное из известных средств описания синтаксиса. Однако эта форма достаточно проста, широко используется и предоставляет достаточные для большинства приложений средства.

На рис. 2 изображена одна из возможных грамматик БНФ для очень узкого подмножества языка Паскаль.

```

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 div 100-a*a;
write(rez,sum)
end.

```

Рис. 1

Грамматика БНФ состоит из множества правил вывода, каждое из которых определяет синтаксис некоторой конструкции языка. Рассмотрим, например, правило 8 на рис. 2:

<присваивание> → ид := <арифметическое выражение>

1. <программа> → <имя программы> var <раздел переменных> begin <раздел операторов> end.
 2. <имя программы> → ид
 3. <раздел переменных> → <список переменных>:<тип>
 4. <список переменных> → ид /<список переменных>, ид
 5. <тип> → integer
 6. <раздел операторов> → <оператор>/<раздел операторов>;<оператор>
 7. <оператор> → <присваивание>/<ввод>/<вывод>/<цикл>
 8. <присваивание> → ид := <арифметическое выражение>
 9. <арифметическое выражение> → <слагаемое>/<арифметическое выражение> +/ - <слагаемое>
 10. <слагаемое> → <значение> / <слагаемое> */div <значение>
 11. <значение> → ид/ конст/(<арифметическое выражение>)
 12. <ввод> → read (<список переменных>)
 13. <вывод> → write (<список переменных>)
 14. <цикл> → for <выражение цикла> to <тело цикла>
 15. <выражение цикла> → ид :=<арифметическое выражение>
 - do <арифметическое выражение>
 16. <тело цикла> → <оператор>/ begin <раздел операторов>
- end

Рис. 2

Это определение оператора присваивания языка ПАСКАЛЬ, обозначенное как <присваивание>. Символ " \rightarrow " можно читать как "является по определению". С левой стороны от этого символа находится определяемая конструкция языка (в нашем случае <присваивание>), а с правой – описание синтаксиса этой конструкции. Строки символов, заключенные в угловые скобки, называются нетерминальными символами. Они составляют синтаксические классы языка. Символы, не заключённые в угловые скобки, называются терминальными символами. Они составляют лексические классы или алфавит языка. В рассматриваемом примере нетерминальными символами являются <присваивание> и <арифметическое выражение>, а терминальными – ид и $:=$. Таким образом, это правило определяет, что конструкция <присваивание> состоит из идентификатора (терминал ид), символа алфавита $:=$, за которым следует конструкция <арифметическое выражение>. Пробелы для написания грамматики не существенны и вставляются только для наглядности.

Для распознавания нетерминального символа <присваивание> необходимо, чтобы существовало определение для нетерминального символа <арифметическое выражение>. Это определение даётся правилом 9 на рис. 2:

$$\text{<арифметическое выражение>} \rightarrow \text{<слагаемое>/<арифметическое выражение> } +/ - \text{ <слагаемое>}$$

Это правило предполагает две возможности, разделённые символом $/$. Первая состоит в том, что <арифметическое выражение> состоит из одной конструкции <слагаемое>. Другой вариант заключается в том, что <арифметическое выражение> состоит из конструкции <арифметическое выражение>, за которым следует знак $+$ или $-$, за которым следует конструкция <слагаемое>. Это правило является рекурсивным, т.е. конструкция <арифметическое выражение> определяется рекурсивно в терминах себя самой. В соответствии с этим правилом нетерминальным символом <арифметическое выражение> является любая последовательность из одного или более слагаемых, разделённых знаками $+$ или $-$.

1.2. ФОРМАЛЬНЫЕ ОПРЕДЕЛЕНИЯ ГРАММАТИКИ ЯЗЫКА

Прежде, чем приступать к реализации транслирующих программ, необходимо определить некоторые понятия.

Алфавит языка – непустое, конечное множество символов. Например, пусть алфавит языка включает три символа: { a , b , c }.

Предложение – непустое конечное множество символов алфавита. Например, из трёх символов алфавита мы можем получить бесконечное количество предложений: aabb, abcc, bac, bca и т.д.

Грамматика устанавливает правила вывода *синтаксически правильных* предложений из всех возможных. Формально грамматика G определяется как четвёрка объектов: (N, T, P, S) , где N – множество нетерминальных символов, образующее синтаксические классы языка; T – множество терминальных символов, образующее алфавит языка; P – множество правил переписывания (например, в форме Бэкуса–Наура); S – начальный нетерминальный символ грамматики, с которого начинается разбор любого предложения языка.

Языком L над грамматикой G называется множество предложений, состоящих из терминальных символов T , выводимых с помощью правил P , начиная с начального нетерминального символа S .

1.3. КЛАССИФИКАЦИЯ ГРАММАТИК

Одна из классификаций грамматик связана с видом правил переписывания P . По этой классификации грамматики разделяют на регулярные, контекстно-свободные, контекстно-зависимые и без ограничений.

Грамматика называется *регулярной*, если правила переписывания имеют вид: $A \rightarrow cB$ или $A \rightarrow c$, где A, B – нетерминальные символы; c – терминальный символ грамматики.

Пример. $G1 = (\{S\}, \{0,1\}, P = \{S \rightarrow 0S/1S/0/1\}, S)$.

Грамматика называется *контекстно-свободной*, если правила переписывания имеют вид: $A \rightarrow w$, где A – нетерминальный символ, $w \in (N \cup T)$ – объединение нетерминальных и терминальных символов грамматики.

Пример. $G2 = (\{A, Z\}, \{a, *, +, (,)\}, P = \{A \rightarrow Z/A+Z, Z \rightarrow a/Z^*a/(A)\}, A\})$.

Грамматика называется *контекстно-зависимой*, если правила переписывания имеют вид: $k \rightarrow w$, где $k, w \in (N \cup T)$ – объединение нетерминальных и терминальных символов грамматики, причём $|k| \leq |w|$.

Пример. $G3 = (\{S, B, C\}, \{a, b, c\}, P = \{S \rightarrow aSBC/abC, CB \rightarrow BC, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}, S\})$.

И, наконец, грамматика *без ограничений* не содержит никаких ограничений в правилах переписывания, которые, в том числе, допускают переход в пустое множество.

Грамматика большинства языков программирования записывается в терминах контекстно-свободных грамматик.

1.4. СИНТАКСИЧЕСКИЕ ДЕРЕВЬЯ

Результат анализа исходного предложения в терминах грамматических конструкций удобно представлять в виде дерева. Такие деревья принято называть синтаксическими.

На рисунке 3, *a* изображено дерево грамматического разбора для предложения `read(a)` с помощью правил 12 и 4 грамматики, представленной на рис. 2.

На рисунке 3, *б* приведено синтаксическое дерево разбора предложения `rez:=sum div100-a*a`. Это дерево выводится с помощью правил 8, 9, 10, 11 грамматики, представленной на рис. 2.



Рис. 3

Таким образом, рассматривая правила грамматики, начиная с начального, можно построить синтаксическое дерево для всей программы, представленной на рис. 1.

Контрольные вопросы

1. Что такое синтаксис языка?
2. Что такое семантика языка?
3. Что определяет грамматика языка?
4. Что такое формальный язык?
5. Для чего используются синтаксические деревья?
6. Как называется транслятор, у которого функциональное назначение – перевод программы с языка низкого уровня в машинные коды?

7. Какая грамматика имеет следующие 4 компонента:
 - множество терминальных символов;
 - множество нетерминальных символов;
 - множество продукции, каждая из которых состоит из нетерминала, называемого левой частью продукции, стрелки и последовательности токенов и/или нетерминалов, называемых правой частью продукции;
 - указание одного из нетерминальных символов как стартового или начального?

2. ТЕОРИЯ ТРАНСЛЯЦИИ

План:

1. Лексический анализ.
2. Синтаксический анализ.
3. Метод рекурсивного спуска.
4. Метод операторного предшествования.
5. Внутреннее представление программы.
6. Генерация кода.

Ключевые слова: трансляция, синтаксический анализ, постфиксная запись, внутреннее представление в виде четырёрок.

2.1. ЛЕКСИЧЕСКИЙ АНАЛИЗ

На вход компилятора, а следовательно, и лексического анализатора поступает цепочка символов некоторого алфавита. Работа лексического анализатора заключается в том, чтобы сгруппировать определённые 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 др., 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.

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.

Таблица 1

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

Для вышеприведённого примера можно составить кодировочную таблицу (табл. 1). Если распознанная лексема является ключевым словом, разделителем или знаком операции, такая схема кодирования даёт всю необходимую информацию. В случае идентификатора или константы необходимы дополнительные данные (в простейшем случае –

типа 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 2

Строка	Код лексемы	Дополнительные данные	Строка	Код лексемы	Дополнительные данные
1	1		7	7	
	22	1		20	
	12			22	3
2	2			21	
	22	2		12	
	14		8	22	2
	22	3		15	
	14			22	2
	22	4		16	
	14			22	3
	22	5	9	4	
	13			12	
	5		10	22	4
	12			15	
3	3			22	2
4	22	2		19	
	15			23	#100
	12			22	3
5	6			18	
	22	5		22	3
	15			12	
	23	# 1	11	8	
	9			20	
	23	# 100		22	4
	10			14	
6	3			22	2
				21	
			12	4	
				11	

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.

2.2. 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. 4).

Восходящий разбор состоит в том, что, отправляясь от заданной цепочки, пытаются привести её к начальному символу (рис. 5).

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



Рис. 4

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

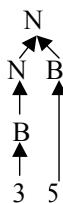


Рис. 5

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

2.3. МЕТОД РЕКУРСИВНОГО СПУСКА

Процесс грамматического разбора для этого метода состоит из отдельных процедур для каждого нетерминального символа, определённого 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 передаёт в вызывающую её 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 $\langle \text{ввод} \rangle$, прежде всего исследует две последовательные лексемы "read" и "(" . В случае совпадения эта процедура вызывает другую процедуру, соответствующую нетерминальному символу $\langle \text{список переменных} \rangle$. Если эта процедура закончится успешно, то процедура $\langle \text{ввод} \rangle$ сравнивает следующую лексему с ")". Если все эти проверки окажутся успешными, то процедура $\langle \text{ввод} \rangle$ завершается с признаком успеха и устанавливает указатель текущей лексемы на лексему, следующую за ")" .

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

<оператор> → <присваивание>/<ввод>/<вывод>/<цикл>

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 столкнёмся 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.

<список переменных> → ид / <список переменных>, 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 приведёт ещё 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:

<список переменных> → ид {, ид}

Эта запись, являющаяся широко принятым расширением БНФ, означает, что конструкция, заключённая 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, разделённых 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 совпадёт с запятой и ид. Такая запись устраняет проблему рекурсии, а также решает вопрос выбора из двух альтернатив.

Грамматика языка, к которому принадлежит предложение (рис. 1), имеет вид, представленный на рис. 6.

```

<программа> → <имя программы> var <раздел переменных>
begin <раздел операторов> end.
<имя программы> → ид
<раздел переменных> → <список переменных>:<тип>
<список переменных> → ид {, ид}
<тип> → integer
<раздел операторов> → <оператор> {;<оператор>}
<оператор> → <присваивание>/<ввод>/<вывод>/<цикл>
<присваивание> → ид := <арифметическое выражение>
    <арифметическое выражение> → <слагаемое>
{+<слагаемое>} {-<слагаемое>}
    <слагаемое> → <значение> {*<значение>} {div <значение>}
    <значение> → ид/ конст/(<арифметическое выражение>)
    <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

```

Рис. 6

Рассмотрим примеры алгоритмов синтаксического анализа методом рекурсивного спуска для некоторых предложений исходной программы с использованием приведённой 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 приведённой 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 рисунке 7 графически представлен процесс грамматического разбора методом рекурсивного спуска для предложения READ. На рисунке 7, *a* изображен вызов процедуры <ввод>, которая обнаружила лексемы READ и) во входном потоке (штриховая линия). На рисунке 7, *б* процедура <ввод> вызывает процедуру <список переменных> (сплошная линия), которая обработала лексему ид. На рисунке 7, *в* процедура <список переменных> закончила работу, передала управление процедуре <ввод> с признаком успешного завершения; процедура <ввод> обработала входную лексему). На этом анализ входного предложения завершён.

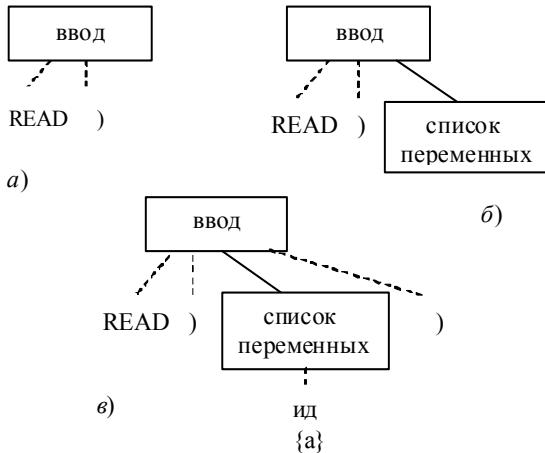


Рис. 7

Приведём ещё 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;
  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 следующим образом:

1. Вызывается процедура <присваивание>, которая обнаружила лексемы ид и := во входном потоке (рис. 8, а).
2. Процедура <присваивание> вызывает процедуру <арифметическое выражение> (рис. 8, б).

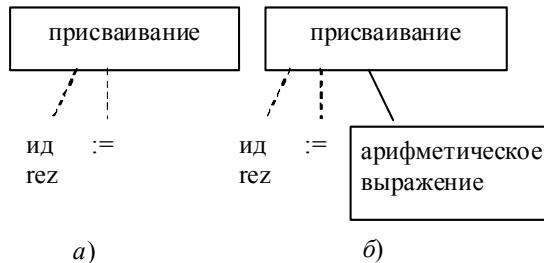


Рис. 8

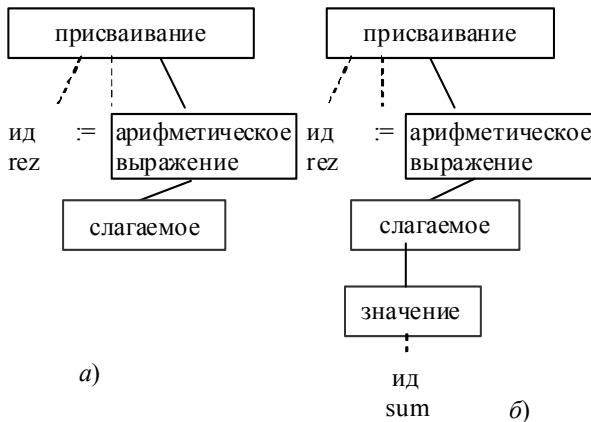


Рис. 9

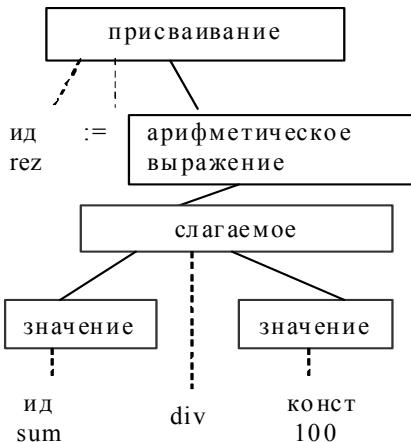
3. Процедура <арифметическое выражение> вызывает процедуру <слагаемое> (рис. 9, а).

4. Процедура <слагаемое>. вызывает процедуру <значение>, которая обнаруживает лексему ид. Управление возвращается в процедуру <слагаемое> (рис. 9, б).

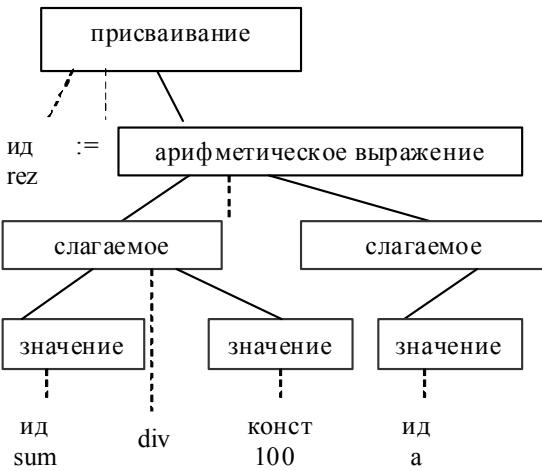
5. Процедура <слагаемое> обнаруживает лексему div и вызывает процедуру <значение>, которая обнаруживает лексему конст и передаёт управление в процедуру <слагаемое>, которая передаёт управление в процедуру <арифметическое выражение> (рис. 10, а).

6. Процедура <арифметическое выражение> распознаёт лексему – , затем вызывает процедуру <слагаемое>, которая вызывает процедуру <значение>, которая распознаёт лексему ид и передаёт управление в процедуру <слагаемое> (рис. 10, б).

7. Процедура <слагаемое> распознаёт лексему * , затем вызывает процедуру <значение>, которая распознаёт лексему ид. Следующая лексема читается в процедуре <значение>. Эта лексема не относится к данному предложению. Управление передаётся в процедуру <слагаемое>, которая формирует признак успешного завершения и передаёт управление в процедуру <арифметическое выражение>. Эта процедура, в свою очередь, успешно заканчивается и передаёт управление в процедуру <присваивание>, которая формирует признак успешного завершения. На этом разбор предложения заканчивается (рис. 11).



a)



б)

Рис. 10

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>`.

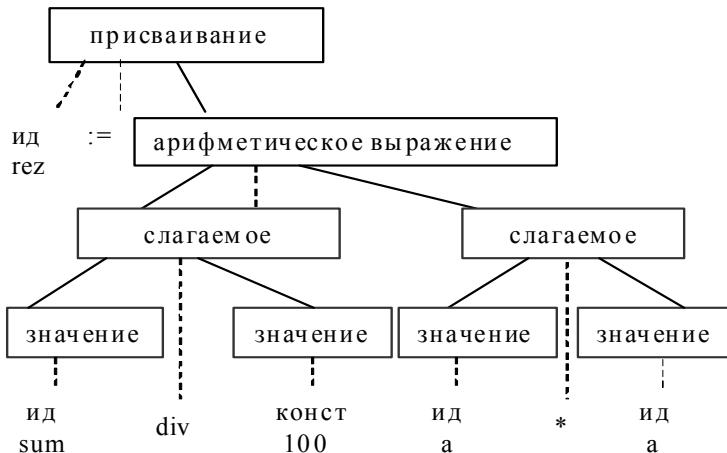


Рис. 11

В результате работы этой процедуры будет построено дерево грамматического разбора для всей программы.

2.4. 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 всё 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:

$$+ < \cdot *$$

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

$* \bullet > -$

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

$A + B^*C - B$
 $\quad <\bullet \quad \bullet>$

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 её в терминах заданной грамматики, до анализа соседних термов предложения.

Предшествующее изложение иллюстрирует основную идею, на которой основан метод грамматического разбора, построенный на отношениях 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 распознаётся в терминах правил вывода используемой грамматики. Этот процесс продолжается до тех пор, пока не будет 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, представленный на рис. 2.

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`, `(`. Приведём матрицу, которая задаёт отношения предшествования для нашей грамматики (табл. 4).

Таблица 4

Лексемы	КОНСТ																									
	ИД																									
program																										
var																										
begin																										
end																										
integer																										
for																										
read																										
write																										
do																										
to																										
integer																										
end.																										
end																										
begin																										
var																										
const																										

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 всё же встретится в процессе грамматического разбора, то она должна рассматриваться как синтаксическая ошибка.

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

Приведём 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 процессор грамматического разбора выделим фрагмент, ограниченный отношениями <• и •> для распознавания в терминах грамматики. В данном случае этот фрагмент содержит един-

ственную лексему ид. Эта лексема может быть распознана как нетерминал <значение> в соответствии с правилом из грамматики. Однако эта лексема может быть также распознана как нетерминальный символ <список переменных>. Для рассматриваемого метода не важно, 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 закончен. Если мы сравним деревья грамматического разбора для этого предложения, то увидим, что они полностью совпадают, за исключением имён 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. 12.

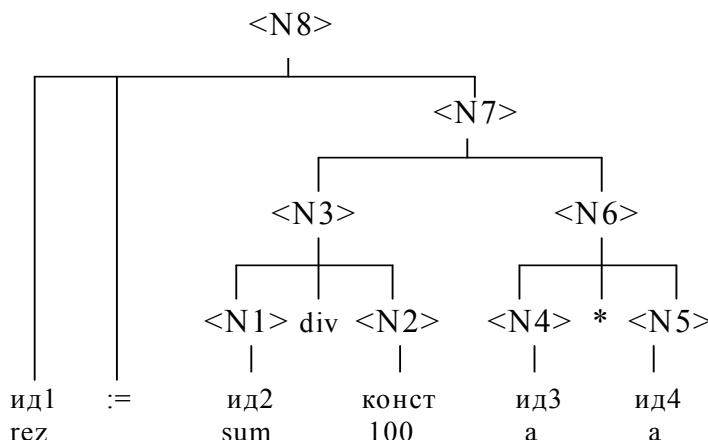


Рис. 12

Заметим ещё 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.

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. 2), то увидим, что оно несколько отличается от полученного методом операторного предшествования дерева (рис. 12).

Например, идентификатор `sum` был сначала интерпретирован как `<значение>`, а потом как `<слагаемое>`, являющийся одним из operandов операции `div`. При разборе методом операторного предшествования идентификатор `sum` был интерпретирован как единственный нетерминал `<N1>`, который является operandом 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 необходимо, поскольку в соответствии с грамматикой (правило 8) первым operandом операции умножения должен быть `<слагаемое>`, а не `<значение>`, так как для нашего метода имена нетерминальных символов несущественны, то подобные переименования становятся ненужными. 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 приведён на рис. 13.

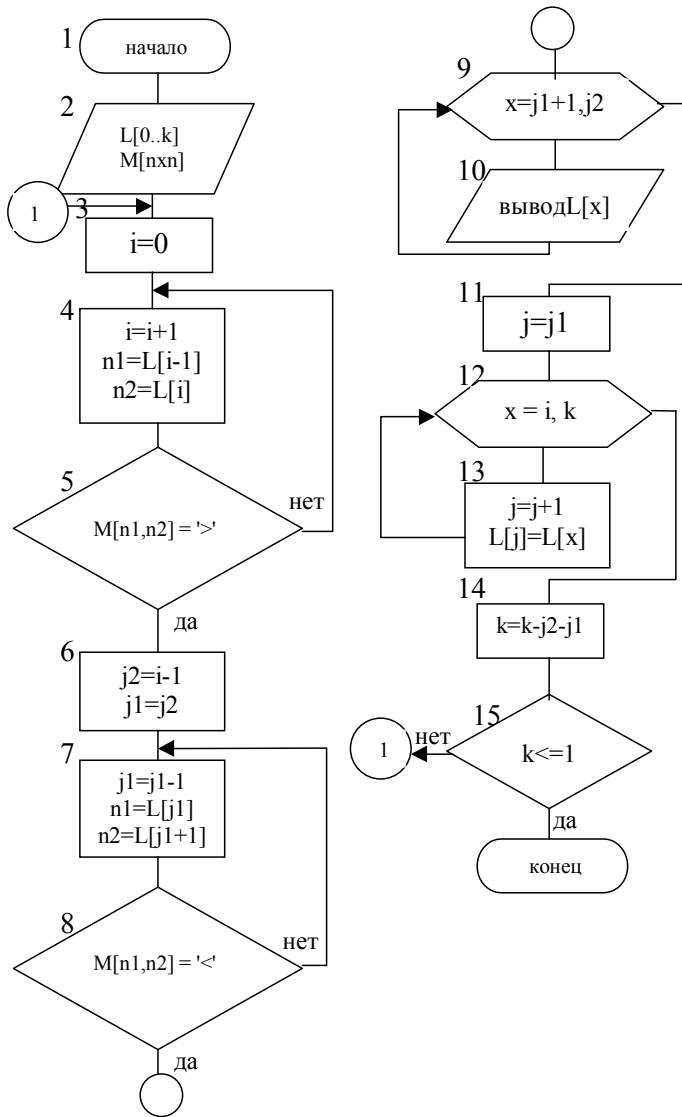


Рис. 13

В данном алгоритме первоначально просматривается цепочка лексем (массив L), устанавливается отношение предшествования между соседними лексемами по таблице отношений предшествования

(матрица $M[nxn]$) до тех пор, пока не встретится отношение '<>' (блоки 4 – 5). После этого возвращаемся по массиву лексем назад, пока между лексемами не встретится отношение '< >', 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.

2.5. 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 четвёрок, последовательность троек, постфиксная запись, префиксная запись, синтаксическое дерево.

1. Последовательность четвёрок

Каждая четвёрка записывается в виде:

операция оп1, оп2, результат,

где операция – выполняемая объектным кодом функция; оп1, оп2 – операнды этой операции; результат определяет, куда должно быть помещено результирующее значение.

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

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

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

Все четвёрки 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 таблице 3 представлена последовательность четвёрок, соответствующая исходной программе.

За операциями read и write следует четвёрка Param, определяющая параметры операций read и write. Четвёрка Param будет, разумеется, при окончательной генерации машинного кода отранслирована в список параметров. Операция > в четвёрке 3 сравнивает значение двух своих operandов и осуществляет переход к четвёрке 9, если первый

Таблица 5

	Операция	Операнд 1	Операнд 2	Результат
№	1	2	3	4
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		gez
13	write			
14	param	rez		
15	param	sum		

операнд больше второго. Операция goto в четвёрке 8 осуществляет безусловный переход к четвёрке 3.

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

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 operandами. Например: $(A + B) * C$.

Вычисление такого выражения является непростой задачей. Операцию умножения нельзя выполнить до тех пор, пока не будет прочитан второй operand C. Если этот operand 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 operandами. Такая запись обладает двумя цennыми свойствами, благодаря которым её 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 дополнительных 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: $A B @ C E * + +$.

2.6. ГЕНЕРАЦИЯ КОДА

Возможны три формы объектного кода: абсолютные команды, помешанные в фиксированные ячейки (после окончания компиляции такая программа немедленно выполняется); программа на автокоде (ее потом придётся 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 для 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 символьические имена. Более того, можно формировать макроопределение. Это позволяет также уменьшить объём компилятора. Несмотря на очевидные достоинства, трансляция на автокод обычно считается наихудшим из вариантов. И в самом деле, к процессу трансляции добавляется ещё один шаг, который часто требует столько же времени, сколько длится собственно компиляция. Боль-

шинство 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 четвёрок и постфиксной записи, используя в качестве примера выражение

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

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

Генерация кода для последовательности четвёрок.

Для рассматриваемого примера последовательность четвёрок 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 четвёрки of

* : подпрограмма, соответствующая операции *;

+ : подпрограмма, соответствующая операции + ;

- : подпрограмма, соответствующая операции – ;

end;

Генерация кода для постфиксной записи. Для рассматриваемого примера постфиксная запись имеет вид:

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

Операторы и operandы просматриваются последовательно слева направо. Всякий раз, когда просматривается operand, в стек заносится его имя, а когда встречается операция, генерируются команды для ее выполнения. При этом в качестве описаний operandов 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 $S(1), S(2), \dots, S(n)$. 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 перемещаться в сторону начала массива (рис. 14).

					i
Вершина					
1	2	3	4	5	

Рис. 14

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.

Контрольные вопросы

1. Какова роль лексического анализатора?
2. Что такое лексемы?
3. Каким этапом компиляции является синтаксический анализ?
4. Какими бывают методы синтаксического анализа?
5. Какой этап предшествует синтаксическому анализу?
6. Что означает данная запись $\langle\text{список переменных}\rangle \rightarrow \text{id}\{\text{id}\}$?
7. Конечными символами грамматического дерева являются ...
8. В каком виде лексический анализатор должен представлять исходный текст?
9. Во время какого этапа предложения распознаются как языковые конструкции используемой грамматики?
10. Этот метод относится к восходящим, которые начинают разбор с конечных узлов грамматического дерева и пытаются объединить их построением узлов всё более и более высокого уровня до тех пор, пока не будет достигнут корень дерева.
11. В методе операторного предшествования переименование нетерминальных символов возможно?
12. Чем отличаются способы нисходящего и восходящего грамматического разбора?
13. Какие известны методы синтаксического анализа?
14. В чём сущность метода рекурсивного спуска?
15. Для чего применяется изменённый способ записи БНФ?

16. В каком виде представляется программа на выходе синтаксического анализатора?
17. В чём сущность метода операторного предшествования?
18. Запись, в которой знак операции ставится непосредственно за операндами, называется...
19. Сколько возможно форм объектного кода?
20. В какой форме объектного кода не приходится формировать команды как последовательности битов; можно порождать команды, содержащие символические имена?
21. Почему программу на автокоде считают наихудшим из вариантов?
22. В генерации кода для постфиксной записи для указания вершины стека обычно используют индекс i . Какое значение i перед чтением стека будет означать, что стек пуст?

3. ОРГАНИЗАЦИЯ ТАБЛИЦ СИМВОЛОВ

План:

1. Способы организации таблиц символов.
2. Неупорядоченные и упорядоченные таблицы.
3. Хеш-адресация.
4. Рехеширование.
5. Метод цепочек.

Ключевые слова: таблица символов, поиск, коллизия.

3.1. СПОСОБЫ ОРГАНИЗАЦИИ 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 ведётся 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		

Таблица 7

№	Аргумент	Значение
0		
1		
...
h(a)		
...
N-1		

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. Сортировка таблицы производится методом упорядоченных вставок.

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). Таблица 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.

Возникает, однако, затруднение, если результаты хеширования двух разных символов совпадают. Это называется коллизией. Очевидно, в данной позиции таблицы может быть помещён 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

удаётся, поэтому разработчику компилятору следует предусмотреть способы решения задачи коллизии. Существуют два таких способа – рехеширование и метод цепочек.

3.2. 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. 15.

Хеш-функция, применённая к символу, даёт 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.
4. Пока поступают символы, хеширование которых даёт индексы разных указателей, они заносятся в таблицу аналогичным образом. Так, если мы записываем в таблицу символы S2, S3, S4, хеширование которых даёт ссылки на указатели 1, 3, 6, таблица примет вид, представленный на рис. 16.

№	Хеш-таблица
0	
1	
...	
N-1	

№	Аргумент	Значение	CHAIN	УК = 0
1				
2				
...				
N				

Рис. 15

№	Хеш-таблица	№	Аргумент	Значение	CHAIN	УК = 4
0		1	S1	...	0	
1	2	2	S2	...	0	
2		3	S3	...	0	
3	3	4	S4	...	0	
4	1					
5						
6	4					

Рис. 16

В конце концов, поступит символ S5, который ссылается на указатель, использовавшийся ранее, например на 6. Вот здесь и начинает действовать поле CHAIN. Символ S5 записывается в таблицу символов и добавляется к концу цепочки для этого указателя (рис. 17).

№	Хеш-таблица	№	Аргумент	Значение	CHAIN	УК = 5
0		1	S1	...	0	
1	2	2	S2	...	0	
2		3	S3	...	0	
3	3	4	S4	...	5	
4	1	5	S5	...	0	
5						
6	4					

Рис. 17

Внесем в таблицу символы S6, S7, S8, которые ссылаются на указатели 4, 3, 3 соответственно (рис. 18).

№	Хеш-таблица	№	Аргумент	Значение	CHAIN	УК = 8
0		1	S1	...	6	
1	2	2	S2	...	0	
2		3	S3	...	7	
3	3	4	S4	...	5	
4	1	5	S5	...	0	
5		6	S6	...	0	
6	4	7	S7	...	8	
7		8	S8	...	0	

Рис. 18

Контрольные вопросы

1. В каком типе организации таблиц символов поиск элемента требует сравнения с каждым элементом таблицы, пока не будет найден подходящий?
2. Сколько (максимум) сравнений бинарный поиск требует для $n = 128$ элементов?
3. Какой метод рехеширования состоит в том, что принимаем $p1 = 1, p2 = 2, p3 = 3$ и т.д.?
4. В этом типе рехеширования принимается $p_i = i * h$, где h – исходный хеш-индекс.
5. Какое поле в методе цепочек используется для того, чтобы связать в цепочку элементы, для которых хеширование символа приводит к тому же самому указателю?
6. Для чего нужна кодировочная таблица?
7. Для чего служит таблица символов?
8. Как организуется таблица символов?
9. В чём состоит метод бинарного поиска?
10. Что такое хеширование?
11. Как работает метод цепочек?
12. Какие известны способы рехеширования?
13. Как сравниваются известные способы организации таблиц символов?
14. В чём суть методов бинарного поиска и упорядоченных вставок?
15. Каковы достоинства хеш-адресации и каковы её недостатки?
16. Какие известны способы рехеширования?
17. В чём преимущество метода цепочек по сравнению с рехешированием?
18. Какая информация хранится в таблице символов?
19. Как записать информацию о переменных?

4. ОПТИМИЗАЦИЯ КОДА

План:

1. Машинно-зависимая оптимизация.
2. Машинно-независимая оптимизация.
3. Методы оптимизации кода.

Ключевые слова: общие подвыражения, инварианты цикла.

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. Алгоритмы и детали, касающиеся этих методов, можно найти в работах [2, 3].

Одним из важных источников оптимизации кода является удаление *общих подвыражений*, которые встречаются в нескольких местах программы и вычисляют одно и то же 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. 19). Следует отметить, что в первоначальном варианте требуется выполнить 161 операцию.

Если мы исследуем эту последовательность четвёрок, то обнаружим, что четвёрки 5 и 11 совпадают, за исключением имени получаемого промежуточного результата. Обратите внимание, что операнд j не меняет своего значения между четвёрками 5 и 11.

1)	$:=$	#1	I	Инициализация цикла
2)	$>$	I	#10	(19)
3)	-	I	#1	i1
4)	*	i1	#10	i2
5)	*	#2	J	i3
6)	-	i3	#1	i4
7)	-	i4	#1	i5
8)	+	i2	i5	i6
9)	-	I	#1	i8
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
16)	$:=$	i14	I	
17)	GOTO		(2)	
18)	...			Следующая операция

Рис. 19

Невозможно достичь четвёрки 11, не проходя предварительно четвёрку 5, поскольку они расположены на одном линейном участке.

Таким образом, четвёрки 5 и 11 вычисляют одно и тоже значение. Это означает, что мы можем удалить четвёрку 11 и заменить любые обращения к её результату ($i9$) на обращение к переменной $i3$, которая является результатом четвёрки 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 четвёрки 6 и 12 также совпадают, за исключением имени результата. Следовательно, мы можем удалить четвёрку 12 и заменить переменную $i11$ всюду, где она используется на переменную $i4$. Аналогично четвёрки 10 и 11 также могут быть удалены, поскольку они эквивалентны четвёркам 3 и 4. В результате получим новую последовательность четвёрок (рис. 20), которая предполагает выполнение всего 121 операции.

Обратите внимание, что общее количество четвёрок сокращено с 17 до 13. Поскольку операции во всех используемых здесь четвёрках 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 называется подвыражение внутри цикла, результирующие значения которых не изменяются внутри цикла при переходе от одной итерации к другой. Таким образом, эти значения могут быть

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. 20

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.

Примечание: необходимо выполнить 94 операции.

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 четвёрку 5 непосредственно перед началом выполнения цикла. Аналогичные соображения относительно четвёрок 6 и 7.

В результате получим новую последовательность четвёрок (рис. 21). Общее количество четвёрок остаётся тем же, но количество четвёрок в цикле уменьшилось с 12 до 9. Каждое выполнение предложения FOR вызывает десятикратное выполнения тела цикла. Это означает, что общее количество операций, необходимых для выполнения FOR, сократилось со 121 до 94.

Общее количество операций, приходящихся 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 ещё более оптимизированный вариант кода.

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. 21

Контрольные вопросы

1. В чём заключается основная цель фазы оптимизации?
2. Какие существуют критерии эффективности для оптимизации кода?
3. Преобразования, осуществляемые на фазе оптимизации, должны приводить к программе, эквивалентной исходной или иной?
4. Что такое машинно-зависимая оптимизация?
5. Что такое машинно-независимая оптимизация?
6. Что такое оптимизация константных вычислений?
7. В чём принцип уменьшения силы операций?
8. Что такое общие подвыражения?
9. Что такое инварианты циклов?

5. ОРГАНИЗАЦИЯ ДИАЛОГА В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

План:

1. Этапы проектирования.
2. Типы диалога.
3. Проектирование диалога типа «меню».
4. Проектирование диалога типа «вопрос–ответ».
5. Проектирование диалога типа «заполнение бланка».
6. Проектирование диалога типа «прямой режим».
7. Средства помощи и поддержки в интерактивной системе.

Ключевые слова: диалоговые вычислительные системы, пользователь, информационная система.

Важнейшим условием правильной эксплуатации вычислительной системы является продуманная организация взаимодействия пользователя и комплекса технических средств.

Часто компьютерную систему используют непрофессиональные пользователи, поэтому некоторые важные их характеристики требуют пристального внимания. По одному из определений, непрофессиональными считаются пользователи, которые обращаются к компьютерным системам в своей повседневной работе или жизни эпизодически. Система должна соответствовать потребностям всех потенциальных пользователей. Существует разница между потребностями тех, кто может случайно воспользоваться системой, и тех, кто может одной системой пользоваться часто, а другой только изредка. Необходимо удовлетворить потребности и пользователя, решющего определённую задачу с помощью интерактивной системы, и научного работника, ис-

пользующего программы, которые он сам написал, и администратора, который использует информацию из базы данных, и бухгалтера, желающего рассчитать экономические показатели, и ещё многих других пользователей.

Часто, из-за стремления ускорить процесс разработки, разработанные системы являются неудобными в работе и не пользуются успехом у пользователей. Удачное внедрение интерактивной системы определяется несколькими факторами:

1. Прежде всего важна окупаемость затраченных средств.
2. В разработанном пакете программ может не оказаться возможностей, которые необходимы для данного конкретного приложения или, наоборот, пользователи могут оказаться недостаточно подготовленными, чтобы в полной мере воспользоваться всеми возможностями системы.
3. Важной является скорость выполнения поставленных перед системой задач.
4. Должна быть высокой наглядность представления данных.
5. Существенно, чтобы новая интерактивная система не вносила путаницы там, где уже есть другие системы, к которым пользователи привыкли. Возможными источниками такой путаницы могут быть:
 - а) использование одинаковых команд (или сокращений) для обозначения различных действий;
 - б) нестандартное использование клавиш специального назначения или других клавиш;
 - в) различие в размещении данных на экране монитора.
6. Система должна быть гибкой, особенно, если она создаётся для долговременного использования. При этом необходимо оговорить, что стоимость системы окажется выше, чем первоначально ожидалось, так как её модернизация потребует дополнительных расходов.
7. Если создаваемая система при работе создаёт собственные системы, то должен осуществляться контроль за ресурсами.

5.1. ПРОЕКТИРОВАНИЕ ИНТЕРАКТИВНЫХ СИСТЕМ

Этапы проектирования интерактивной системы:

- 1) предложение новой системы;
- 2) постановка задачи;
- 3) детальная разработка системы;
- 4) тестирование системы;
- 5) внедрение системы.

Казалось бы, нет никаких различий от любой другой программной системы. Существенная разница кроется в том, что разрабатыва-

мая система непосредственно используется людьми, не являющимися специалистами в области вычислительной техники, и предназначена помочь им в их обычной работе. Поэтому будущих пользователей нужно привлекать к процессу разработки, но они не могут представить проектируемую систему, не зная способа, с помощью которого будет вводиться информация и выводиться конечный результат. Другими словами, тип диалога в интерактивной системе должен быть обозначен в общих чертах на самой ранней стадии разработки. На практике наиболее подходящий тип выбирается с учётом целей диалога и вида пользователей.

Перечислим различные типы диалога:

- выбор из меню;
- вопрос–ответ;
- заполнение бланков:
 - с произвольным порядком заполнения;
 - с указанием порядка заполнения автоматическим перемещением курсора;
- "прямой режим" с использованием языка команд.

При организации диалога используются различные режимы вывода информации на экран монитора:

- режим "весь экран" (при этом после каждого ответа изменяется вся информация на экране или добавляется информация в определённое место экрана);
- режим "строка за строкой" (на экран выводится одна строка и после каждого шага диалога одна строка изменяется на другую, при этом вся информация на экране стирается и новая строка появляется в определённом месте экрана);
- переход от одной страницы к другой (очередная вводимая или выводимая строка появляется на экране внизу и одновременно теряется одна строка вверху; недостаток этого варианта в том, что информация на экране кажется непрерывно движущейся, так что данные трудно рассмотреть; с другой стороны, всегда видна последняя часть диалога);
- редактирование содержимого экрана (это своего рода экранные редакторы, позволяющие изменять информацию на экране).

5.2. ДИАЛОГ НА ОСНОВЕ ВЫБОРА ИЗ МЕНЮ

В такой системе пользователю предоставляется список вариантов выбора, возможных в данном месте диалога. Выбор приводит к появлению на экране либо последующих меню, либо к появлению требуе-

мой информации, либо к переводу терминала в состояние, в котором можно вводить данные.

Общепринято, что системы, основанные на выборе меню, являются подходящими для новичков и случайных пользователей по двум причинам:

1) обычно наряду со списком действий, которые нужно предпринять, имеется достаточночное число инструкций;

2) одно меню похоже на другое и это придаёт уверенности пользователю.

Но такие системы не лишены и недостатков:

1) такой тип диалога может показаться скучным для опытных пользователей;

2) они требуют больше компьютерных ресурсов, чем системы, использующие менее многословные типы диалога.

Режим "весь экран" больше подходит для меню, так как он допускает широкое использование инструкций и других средств помощи пользователю.

Применение меню уменьшает число символов, которые печатаются пользователем; ввод, таким образом, становится совершенно незатруднительным даже для неопытных работников. Ввод может быть упрощён до такой степени, что не всегда требуется клавиатура в полном объёме.

Не следует забывать, что средства, которые помогают пользователю, такие как указание выбора или напоминание о возможностях выбора на очередном шаге, могут действовать раздражающе на более опытного специалиста. Если система должна обслуживать и тех и других, можно предусмотреть альтернативные варианты. Это вызывает дополнительные затраты и необходимо заранее решить, окупятся ли они.

Меню могут существовать внутри других типов диалога. В частности, они могут появляться внутри диалога "вопрос–ответ" в момент, когда нужно перейти по одному из нескольких направлений. Это позволит сократить число вопросов.

5.3. ДИАЛОГ ТИПА "ВОПРОС–ОТВЕТ"

Диалог "вопрос–ответ" ведётся точно так, как подразумевает название: система задаёт вопрос, пользователь даёт ответ.

Достоинства такого типа диалога:

- 1) пригоден для неопытных пользователей;
- 2) лёгок при обучении.

Не лишен такой диалог и недостатков:

- 1) неудобен, если требуется неоднозначный или обширный ответ;

2) диалог должен соответствовать каждому конкретному приложению;

3) интерактивная система с таким диалогом лишена гибкости, так как структура диалога зафиксирована.

При проектировании вопросно-ответных систем необходимо:

1) чётко определить логическую структуру диалога;

2) исключить возможность повторяющихся вопросов;

3) чётко сформулировать вопросы, так, чтобы они допускали только однозначные ответы;

4) исключить лишние вопросы, не имеющие отношения к теме диалога;

5) продумать логичное размещение вопросов на экране;

6) продумать возможность вывода и даже редактирования протокола диалога.

5.4. ДИАЛОГ ТИПА "ЗАПОЛНЕНИЕ БЛАНКА"

Такой диалог предполагает, что на экране появляется так называемый "бланк". Пользователь вводит сообщение в поля бланка.

При проектировании такого типа диалога необходимо:

1) выбрать способ перемещения курсора по полям бланка:

– автоматическое перемещение (после окончания ввода поля курсор перемещается в другое поле по заранее заданному маршруту);

– произвольное перемещение (пользователь сам задаёт маршрут перемещения курсора);

2) продумать расположение полей на экране, в том числе и их логический порядок;

3) оценить количество информации и предусмотреть, если необходимо, переход от одного окна к другому.

5.5. ДИАЛОГ ТИПА "ПРЯМОЙ РЕЖИМ"

Такой тип диалога предполагает наличие языка команд. Для его организации пользователь вводит некоторую команду с необходимыми параметрами и ключами.

При проектировании диалога типа "прямой режим" необходимо:

1) продумать грамматику языка команд;

2) предусмотреть, если возможно, использование символов одного регистра в одной команде;

3) предусмотреть возможность сокращения названия команд и их параметров;

4) предусмотреть однотипность названия команд, чтобы не возникало сложностей при переходе от одной команды к другой.

5.6. СРЕДСТВА ПОМОЩИ И ПОДДЕРЖКИ

Любая интерактивная система должна предоставлять пользователю возможность получить помощь, а при необходимости и поддержку. Поэтому разработчик вычислительной системы должен заранее предусмотреть способы их организации:

- 1) при вводе информации в диалоге необходимо провести проверку достоверности вводимых данных;
- 2) выводить сообщения о допущенных в процессе диалога ошибках;
- 3) при необходимости предусмотреть возможность вызова помощи. Для этого можно:
 - нажать функциональную клавишу (например, F1);
 - ввести определённые символы (например, HELP);
 - использовать специальное поле для ввода символа помощи;
- 4) предусмотреть поддержку:
 - неодушевлённую (документы, справочники, специальную литературу);
 - одушевлённую (специалист, который сопровождает систему).

Контрольные вопросы

1. Каковы факторы успешного внедрения интерактивных систем?
2. Кто является непрофессиональным пользователем?
3. Какие типы диалога существуют?
4. Какие типы диалога лучше подходят для непрофессиональных пользователей?
5. Какие типы диалога лучше подходят для профессиональных пользователей?
6. Для чего нужны средства помощи в интерактивных системах?
7. Для чего нужны средства поддержки в интерактивных системах?

УПРАЖНЕНИЯ

1. Записать грамматику и построить синтаксическое дерево для заданного предложения.
2. Привести алгоритм лексического анализа для заданного предложения.
3. Привести алгоритм синтаксического анализа методом рекурсивного спуска для заданного предложения.
4. Представить матрицу отношений предшествования для заданного предложения.
5. Привести алгоритм синтаксического анализа методом операторного предшествования для заданного предложения.

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.0) T=-B/(2*A)
TYPE*, T
4. S1:=A MOD B;
S2:=A-((A DIV B)*B);
if S1=S2 then write ('BCĒ')
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.0) Y=X*X
if (X.GT.0) 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;
writeln (A)
12. A:=-3*C;
B:=Y*Y*X;
if (A<0) AND (B>0) THEN
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 ('бepho')
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 $A\leq B$ 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 "ЛИНГВИСТИЧЕСКИЕ СРЕДСТВА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ"

Основная цель – закрепление навыков создания лингвистического обеспечения, 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 "Лингвистические средства вычислительных систем" для бакалавров 2 курса специальности 230100 следующие.

В отчёте по курсовой работе необходимо отразить 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, её 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).

Взаимодействие между компонентами компилятора может осуществляться разными способами. На рис. 22, а показано, что лексический анализатор (ЛА) считывает исходную программу (ИП) и представляет её в виде файла лексем. Синтаксический анализатор (СА) читает этот файл и выдаёт внутреннее представление (ВП) программы. Наконец, этот файл считывается генератором кода (ГК), который создаёт объектный код программы. Компилятор такого вида называется трёхпроходным, так программа считывается трижды (исходная программа, лексемы, внутреннее представление).

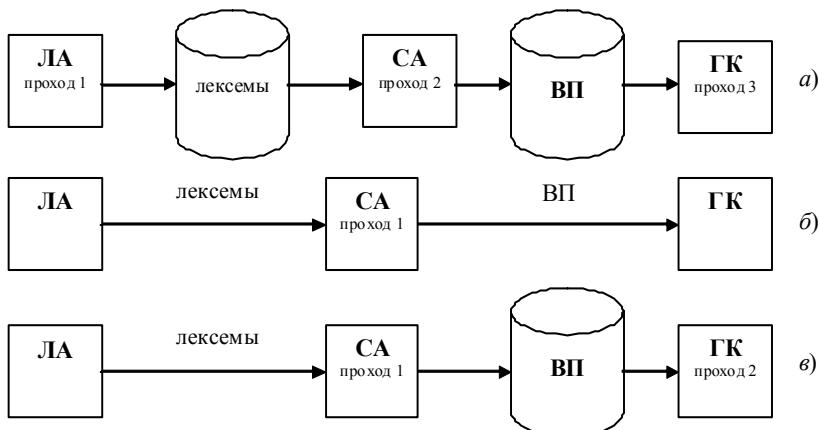


Рис. 22. Способы организации трансляторов

На рисунке 22, 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 создаёт объектный код для этого фрагмента программы.

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

Промежуточное положение между описанными двумя вариантами занимает двухпроходный компилятор (рис. 22, в). В этом случае синтаксический анализатор, вызывая лексический анализатор, получает лексемы и строит файл во внутреннем представлении. Генератор кода считывает этот файл и создаёт объектный код.

ВАРИАНТЫ ЗАДАНИЙ

1. Разработать однопроходный транслятор с исходного языка на язык Паскаль:

программа B;
переменные с, 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.

2. 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.

3. Разработать трёхпроходный 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.

4. Разработать однопроходный транслятор с исходного языка на язык Ассемблер:

программа C;
переменные x,y,z:целые;
ввод x,y;
если x>y то z=x-y
иначе
z=y-x+1;
вывод z;
конец.

5. Разработать двухпроходный транслятор с исходного языка на язык Ассемблер:

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

6. Разработать трёхпроходный транслятор с исходного языка на язык Ассемблер:

```
program K;
  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
```

7. Разработать однопроходный транслятор с исходного языка на язык Бейсик:

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

8. 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
```

9. Разработать трёхпроходный 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
```

10. Разработать трёхпроходный транслятор с исходного языка на язык Фортран:

программа Р;
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.

11. Разработать двухпроходный транслятор с исходного языка на язык Паскаль:

программа Е;
переменные h: вещественные;
n: целые;
h:=0;
читать(n);
пока $n > 0$ выполнить
 $(h:=h+l/n; n:=n-l)$;
печатать (h)
конец.

12. 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
```

13. Разработать однопроходный транслятор с исходного языка на язык Фортран:

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

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

программа РС;
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.

15. 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  
i. continue  
type *,s  
stop  
end
```

16. Разработать трёхпроходный 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 1 выполнить  
    s2:=s2+b;
```

```
]  
передача (s1)  
конец.
```

17. 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.
```

18. Разработать трёхпроходный 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);
}
}
```

19. 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.
```

20. Разработать однопроходный транслятор с исходного языка на язык Си:

```
программа АВ;
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.
```

21. 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.
```

22. Разработать трёхпроходный 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.
```

23. 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
```

24. 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
```

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

26. 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.
```

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

```
программа c;
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.
```

28. Разработать трёхпроходный 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)
}

```

29. 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.

30. 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
```

31. Разработать трёхпроходный 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
```

32. 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.
```

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

Количество идентификаторов, операторов, порядок их следования могут быть любыми.

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 дисциплин «Лингвистические средства вычислительных систем» и «Теория языков программирования и методы трансляции» бакалавры и магистры изучают вопросы, посвящённые отдельным этапам трансляции. Все необходимые материалы представлены в данной работе. Приведены основные положения теории формальных грамматик и языков. Рассмотрены основные подходы к созданию транслирующих программ. Показаны основные способы организации трансляторов: однопроходных, двухпроходных, трёхпроходных. Приведены методы организации таблиц символов и их взаимосвязь с основными частями транслятора: лексическим анализатором, синтаксическим анализатором и генератором кода. Рассмотрены основные методы машинно-независимой оптимизации, и на их примере – организация внутреннего представления программы, в том числе циклических процедур.

СПИСОК ЛИТЕРАТУРЫ

1. **Ахо, А.** Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – Москва : Издательский дом "Вильямс", 2001.
2. **Ахо, А.** Теория синтаксического анализа, перевода и компиляции / А. Ахо, Дж. Ульман. – Москва : Мир, 1976. – Т. 1, 2.
3. **Бек, Л.** Введение в системное программирование / Л. Бек. – Москва : Мир, 1988.
4. **Грис, Д.** Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – Москва : Мир, 1975.
5. **Маккиман, У.** Генератор компиляторов / У. Маккиман, Дж. Хорнинг, Д. Уортман. – Москва : Статистика, 1980.
6. **Рейуорд-Смит, В. Дж.** Теория формальных языков. Вводный курс / В. Дж. Рейуорд-Смит. – Москва : Радио и связь, 1988.
7. **Разработка** компиляторов : лабор. работы / сост. И. Л. Коробова. – Тамбов : Тамб. гос. техн. ун-т, 1997. – 28 с.
8. **Проектирование** трансляторов : метод. указания / сост. : И. Л. Коробова, Д. В. Абрамов. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2000. – 28 с.
9. **Коробова, И. Л.** Основы разработки трансляторов в САПР : учебное пособие / И. Л. Коробова, И. А. Дьяков, Ю. В. Литовка. – Тамбов : Изд-во ТГТУ, 2007. – 80 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1. ТЕОРИЯ ФОРМАЛЬНЫХ ГРАММАТИК И ЯЗЫКОВ	4
1.1. Грамматика	4
1.2. Формальные определения грамматики языка	6
1.3. Классификация грамматик	7
1.4. Синтаксические деревья	8
Контрольные вопросы	8
2. ТЕОРИЯ ТРАНСЛЯЦИИ	9
2.1. Лексический анализ	9
2.2. Синтаксический анализ	12
2.3. Метод рекурсивного спуска	13
2.4. Метод операторного предшествования	22
2.5. Внутреннее представление программы	29
2.6. Генерация кода	31
Контрольные вопросы	33
3. ОРГАНИЗАЦИЯ ТАБЛИЦ СИМВОЛОВ	34
3.1. Способы организации таблиц символов	34
3.2. Метод цепочек	36
Контрольные вопросы	38
4. ОПТИМИЗАЦИЯ КОДА	38
Контрольные вопросы	42
5. ОРГАНИЗАЦИЯ ДИАЛОГА В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ	42
5.1. Проектирование интерактивных систем	43
5.2. Диалог на основе выбора из меню	44
5.3. Диалог типа "вопрос–ответ"	45
5.4. Диалог типа "заполнение бланка"	46
5.5. Диалог типа "прямой режим"	46
5.6. Средства помощи и поддержки	47
Контрольные вопросы	47
УПРАЖНЕНИЯ	47
КУРСОВАЯ РАБОТА ПО ДИСЦИПЛИНЕ "ЛИНГВИСТИЧЕСКИЕ СРЕДСТВА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ"	49
ЗАКЛЮЧЕНИЕ	62
СПИСОК ЛИТЕРАТУРЫ	62
	63

Учебное электронное издание

КОРОБОВА Ирина Львовна

ТЕОРИЯ ТРАНСЛЯЦИИ

Методические указания

Редактор Т. М. Глинкина

Инженер по компьютерному макетированию И. В. Евсеева

Подписано к изданию 16.04.2014
Заказ № 193

Издательско-полиграфический центр ФГБОУ ВПО "ТГТУ"
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Телефон (4752) 63-81-08
E-mail: izdatelstvo@admin.tstu.ru

