МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

О.В. Караваева

ОСНОВНЫЕ ФУНКЦИИ КОМПИЛЯТОРОВ

Лабораторный практикум

Специальность 220100

Киров 2004

Печатается по решению редакционно–издательского совета
Вятского государственного университета

УДК 681.3
К21

Рецензент: доктор физико-математических наук, заведующий кафедрой "Прикладная математика и информатика" А.Н.Рапопорт.

Караваева О.В. Основные функции компиляторов: Лабораторный практикум. - Киров: Изд-во ВятГУ, 2004.- 32 с.

Редактор Е.Г. Козвонина

Подписано в печать | Усл. печ. л. 2
Бумага офсетная | Печать копир Aficio 1022
Заказ № 287 | Тираж 62

Текст напечатан с оригинала-макета, представленного автором.

610000, г.Киров, ул.Московская, 36.

Оформление обложки, изготовление - ПРИП ВятГУ.

Содержание:

ВВЕДЕНИЕ

В настоящее время искусственные языки, использующие для описания предметной области текстовое представление, широко применяются не только в программировании, но и в других областях. С их помощью описывается структура всевозможных документов, трехмерных виртуальных миров, графических интерфейсов пользователя и многих других объектов, используемых в моделях и в реальном мире. Для того чтобы эти текстовые описания были корректно составлены, а затем правильно распознаны и интерпретированы, используются специальные методы их анализа и преобразования. В основе методов лежит теория языков и формальных грамматик, а также теория автоматов. Программные системы, предназначенные для анализа и интерпретации текстов, называются трансляторами.

Транслятор - *обслуживающая программа, преобразующая исходную программу, предоставленную на входном языке программирования, в рабочую программу, представленную на объектном языке*.

Приведенное определение относится ко всем разновидностям транслирующих программ. Однако у каждой из таких программ могут быть свои особенности по организации процесса трансляции. В настоящее время трансляторы разделяются на три основные группы: ассемблеры, компиляторы и интерпретаторы.

Ассемблер - *системная обслуживающая программа, которая преобразует символические конструкции в команды машинного языка*.

Компилятор - *это обслуживающая программа, выполняющая трансляцию на машинный язык программы, записанной на исходном языке программирования*. Так же, как и ассемблер, компилятор обеспечивает преобразование программы с одного языка на другой (чаще всего в язык конкретного компьютера). Вместе с тем команды исходного языка значительно отличаются по организации и мощности от команд машинного языка. Существуют языки, в которых одна команда исходного языка транслируется в 7-10 машинных команд. Однако есть и такие языки, в которых каждой команде может соответствовать 100 и более машинных команд (например, Пролог). Кроме того, в исходных языках достаточно часто используется строгая типизация данных, осуществляемая через их предварительное описание. Программирование может опираться не на кодирование алгоритма, а на тщательное обдумывание структур данных или

классов. Процесс трансляции с таких языков обычно называется компиляцией, а исходные языки обычно относятся к языкам программирования высокого уровня (или высокоуровневым языкам). Абстрагирование языка программирования от системы команд компьютера привело к независимому созданию самых разнообразных языков, ориентированных на решение конкретных задач. Появились языки для научных расчетов, экономических расчетов, доступа к базам данных и другие.

Интерпретатор - *программа или устройство, осуществляющее пооператорную трансляцию и выполнение исходной программы*. В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Как в компиляторах, так и в интерпретаторах используются одинаковые методы анализа исходного текста программы. Но интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки программ более гибким.

Практически во всех трансляторах (и в компиляторах, и в интерпретаторах) в том или ином виде присутствует большая часть перечисленных ниже процессов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерация внутреннего представления программы;
- оптимизация;
- генерация объектной программы.

В конкретных компиляторах порядок этих процессов может быть несколько иным, некоторые из них могут объединяться в одну фазу, другие могут выполняться в течение всего процесса компиляции.

В данном лабораторном практикуме рассмотрены операции, необходимые для компиляции программ типичного языка высокого уровня. Лабораторная работа выполняется с помощью программной модели. Инструкции по использованию модели находятся в файле COMPILE.chm. В качестве примера используется программа на языке Паскаль, изображенная на рисунке 1.1.

```
1       PROGRAM stats;
2       VAR
3            sum, sumsq, i, value, mean, variance: INTEGER;
4       BEGIN
5         sum := 0;
6         sumsq := 0;
7         FOR i:=1 TO 100 DO
8                 BEGIN
9                       READ (value);
10                      sum := sum + value;
11                      sumsq := sumsq + value * value;
12                END;
13        mean := sum DIV 100;
14        variance := sumsq DIV 100 – mean * mean;
15        WRITE (mean, variance);
16      END.
```

Рисунок 1.1 - Пример программы на языке Паскаль

1. Грамматики

Грамматика языка программирования является формальным описанием его синтаксиса или формы, в которой записаны отдельные предложения программы или вся программа.

Описание языков программирования во многом опирается на теорию формальных языков. Эта теория является фундаментом для организации синтаксического анализа и перевода.

Существует два основных, тесно связанных способа определения языков:

- механизм порождения или генератор;
- механизм распознавания или распознаватель.

Первый обычно используется для описания языков, а второй для их реализации. Оба способа позволяют описать языки конечным образом, несмотря на бесконечное число порождаемых ими цепочек.

Неформально язык **L** - это множество цепочек конечной длины в алфавите **V**. Механизм порождения позволяет описать языки с помощью системы правил, называемой грамматикой. Цепочки (предложения) языка строятся в соответствии с этими правилами. Достоинство определения языка с помощью грамматик в том, что операции, производимые в ходе синтаксического анализа и перевода, можно делать проще, если воспользоваться структурой, предписываемой цепочкам с помощью этих грамматик.

Синтаксис - *совокупность правил некоторого языка, определяющих формирование его элементов. Иначе говоря, это совокупность правил образования семантически значимых последовательностей символов в данном языке. Синтаксис задается с помощью правил, которые описывают понятия некоторого языка*. Примерами понятий являются: переменная, выражение, оператор, процедура. Последовательность понятий и их допустимое использование в правилах определяет синтаксически правильные структуры, образующие программы.

Одной из наиболее простых и широко используемых форм записи грамматик является нормальная форма Бэкуса-Наура (БНФ). Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

– символ "::=" отделяет левую часть правила от правой (символ "::=" можно читать как “является по определению”, иногда вместо "::=" используется символ "→");

– нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">" (нетерминалы являются именами конструкций, определенными внутри грамматики);

– терминалы - это символы, используемые в описываемом языке;

– каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты "|".

На рисунке 1.2 изображена одна из возможных грамматик БНФ для очень узкого подмножества языка Паскаль.

1	<prog>	::=	PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2	<prog-name>	::=	**id**
3	<dec-list>	::=	<dec> | <dec-list> ; <dec>
4	<dec>	::=	<id-list> : <type>
5	<type>	::=	INTEGER
6	<id-list>	::=	**id** | <id-list> , **id**
7	**<**stmt-list>	::=	**<**stmt**>**
8	**<**stmt>	::=	<assign>
9	<assign>	::=	id :=<exp>
10	<exp>	::=	<term> | <exp> + <term> | <exp> **–** <term>
11	<term>	::=	<factor> | <term> *<factor> | <term> DIV <factor>
12	<factor>	::=	**id** | **int** | (<exp>)
13	<read>	::=	READ (<id-list>)
14	<write>	::=	WRITE (<id-list>)
15	<for>	::=	FOR <index-exp> DO <body>
16	<index-exp>	::=	id := <exp> TO <exp>
17	<body>	::=	**<**stmt**>** | BEGIN <stmt-list> END

Рисунок 1.2 - Упрощенная грамматика языка Паскаль

Результат анализа исходного предложения в терминах грамматических конструкций удобно представлять в виде дерева, которое называется деревом грамматического разбора или синтаксическим деревом.

Правило вывода 9 на рисунке 1.2 дает определение синтаксиса предложения присваивания: <assign> ::= id := <exp>
Это означает, что конструкция <assign> состоит из лексемы id, за которой следует лексема :=, за которой идет конструкция <exp>. Правило 10 дает определение конструкции <exp> как состоящей из любой последовательности конструкций <term>, соединенных операторами «плюс» или «минус». Аналогично правило 11 определяет конструкцию <term> как последовательность конструкций <factor>, разделенными знаками * или DIV. В соответствии с правилом 12 конструкция <factor> может состоять из идентификатора **id** или целого **int**, которое также распознается сканером, или из конструкции <exp>, заключенной в круглые скобки.

На рисунке 1.3 изображено дерево грамматического разбора для предложения 14 на рисунке 1.1, основанное на только что рассмотренных правилах вывода. В соответствии с этим правилом умножение и деление производятся перед сложением и вычитанием. Прежде всего должны вычисляться термы SUMSQ DIV 100 и MEAN*MEAN, поскольку эти промежуточные результаты являются операндами (левым и правым поддеревом) операции

вычитания. Операции умножения и деления имеют более высокий ранг, чем операции сложения и вычитания. **Такое ранжирование является следствием того, как записаны правила 10-12.** Дерево на рисунке 1.3 представляет собой единственно возможный результат анализа предложения 14 в терминах грамматики на рисунке 1.2. Для некоторых грамматик подобной единственности может не существовать. Если для одного и того же предложения можно построить несколько различных деревьев грамматического разбора, то соответствующая грамматика называется неоднозначной. При разработке компиляторов предпочитают пользоваться однозначными грамматиками.

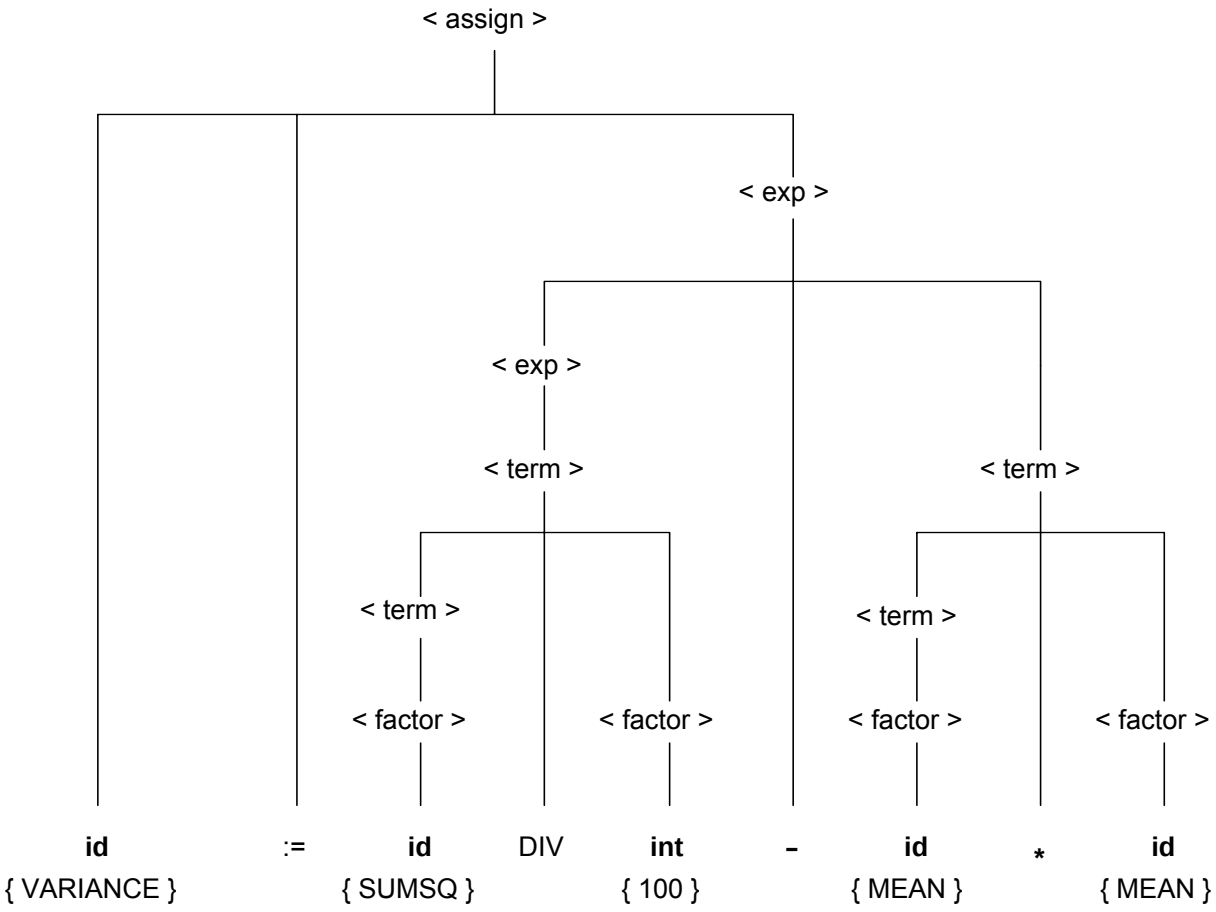


Рисунок 1.3 - Дерево грамматического разбора для предложения 14 программы на рисунке 1.1

2. Лексический анализ

Лексический анализ включает в себя просмотр компилируемой программы и распознавание лексем, составляющих предложения исходного текста. Лексические анализаторы строятся таким образом, чтобы они могли распознавать ключевые слова, операторы и идентификаторы так же, как целые числа, числа с плавающей точкой, строки символов и другие аналогичные конструкции, встречающиеся в исходной программе. Точный перечень лексем, которые необходимо распознать, зависит от языка программирования, на который рассчитан компилятор, и от грамматики, используемой для описания этого языка.

Лексический анализатор преобразует исходную программу в последовательность символов. При этом идентификаторы и константы произвольной длины заменяются символами фиксированной длины. Слова языка также заменяются каким-нибудь стандартным представлением. Например, слова языка могут заменяться целыми числами, идентификаторы – буквой I и следующим за ней целым числом, константы – буквой C и целым числом и т.д. Эти целые числа не могут быть произвольной длины и поэтому число идентификаторов и констант ограниченно какой-либо, обычно очень большой, величиной. Пользователю практически невозможно увидеть данное ограничение.

Лексема	Код
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
.	14
:=	15
+	16
-	17
*	18
DIV	19
(	20
)	21
id	22
int	23

Рисунок 2.1 - Коды лексем для грамматики на рисунке 1.2

Для повышения эффективности последующих действий каждая лексема обычно определяется кодом, например целым числом, а не в виде строки символов переменной длины. Если распознанная лексема является ключевым словом или оператором, такая схема кодирования дает всю необходимую информацию.

Коды, создаваемые для слов языка, не должны зависеть от программы, например: ключевое слово **begin** должно всегда иметь какой-либо конкретный код. В случае же идентификатора дополнительно необходимо конкретное имя распознаваемого идентификатора.

Получение кодов идентификаторов производиться последовательно, начиная с начала по тексту программы. Например, первый идентификатор будет иметь код I1, а второй - I2 и т.д. Каждый экземпляр определенного идентификатора (на любом уровне) заменяется одним и тем же кодом. Из этого следует необходимость создания таблицы идентификаторов, где будут храниться соответствия кодов и самих идентификаторов.

В таблице 2.1 показан результат обработки сканером программы, приведенной на рисунке 1.1 с использованием кодировки лексем, представленной на рисунке 2.1. Для лексем типа 22 (идентификаторы) и типа 23 (целые числа) должны быть определены спецификаторы для указания на таблицу символов и таблицу констант.

Таблица 2.1 – Результат лексического разбора программы на рисунке 1.1

Тип лексемы	Индекс	Символ	Специ-фикатор	Тип лексемы	Индекс	Символ	Специ-фикатор
T	1	PROGRAM		I	22	SUM	#SUM
I	22	STATS	#ST	T	15	:=	
T	2	VAR		I	22	SUM	#SUM
I	22	SUM	#SUM	T	16	+	
T	14	,		I	22	VALUE	#VALUE
I	22	SUMSQ	#SUMSQ	T	12	;	
T	14	,		I	22	SUMSQ	#SUMSQ
I	22	I	#I	T	15	:=	
T	14	,		I	22	SUMSQ	#SUMSQ
I	22	VALUE	#VALUE	T	16	+	
T	14	,		I	22	VALUE	#VALUE
I	22	MEAN	#MEAN	T	18	*	
T	14	,		I	22	VALUE	#VALUE
I	22	VARIANCE	#VARI	T	4	END	
T	13	:		T	12	;	
T	6	INTEGER		I	22	MEAN	#MEAN
T	3	BEGIN		T	15	:=	
I	22	SUM	#SUM	I	22	SUM	#SUM
T	15	:=		T	19	DIV	
C	23	0	^0	C	23	100	
T	12	;		T	12	;	
I	22	SUMSQ	#SUMSQ	I	22	VARIANCE	#VARI
T	15	:=		T	15	:=	
C	23	0	^0	I	22	SUMSQ	
T	12	;		T	19	DIV	
T	7	FOR		C	23	100	
I	22	I	#I	T	17	-	
T	15	:=		I	22	MEAN	#MEAN
C	23	1	^1	T	18	*	

Окончание таблицы 2.1

T	10	TO		I	22	MEAN	#MEAN
C	23	100	^100	T	12	;	
T	11	DO		T	9	WRITE	
T	3	BEGIN		T	20	(	
T	8	READ		I	22	MEAN	#MEAN
T	20	(		T	15	,	
I	22	VALUE	#VALUE	I	22	VARIANCE	#VARI
T	21	)		T	21	)	
T	12	;		T	5	END.	

3 Синтаксический анализ

Синтаксический разбор - это основная часть компилятора на этапе анализа. Она выполняет выделение синтаксических конструкций в тексте исходной программы, обработанной лексическим анализатором. На этой же фазе компиляции проверяется синтаксическая правильность программы.

Синтаксический анализ используется для доказательства того, принадлежит ли анализируемая входная цепочка множеству цепочек, порождаемых грамматикой данного языка. Выполнение синтаксического разбора осуществляется распознавателями, являющимися автоматами. Поэтому данный процесс также называется распознаванием входной цепочки. Результатом работы распознавателя грамматики входного языка является последовательность правил грамматики, примененных для построения входной цепочки.

Выделяются два основных метода синтаксического разбора:

- восходящий разбор;
- нисходящий разбор.

Кроме этого можно использовать комбинированный разбор, сочетающий особенности двух предыдущих.

При **восходящем разборе** дерево начинает строиться от терминальных листьев путем подстановки правил, применимых к входной цепочке в произвольном порядке. На следующем шаге новые узлы полученных поддеревьев используются как листья во вновь применяемых правилах. Процесс построения дерева разбора завершается, когда все символы входной цепочки будут являться листьями дерева, корнем которого окажется начальный нетерминал. Если, в результате полного перебора всех возможных правил, невозможно построить

требуемое дерево разбора, то рассматриваемая входная цепочка не принадлежит данному языку.

При использовании восходящего метода на множестве терминальных и нетерминальных символов необходимо ввести три отношения, называемые "отношения предшествования" и обозначаемые как ⋖, ⋗ и ≐.

Отношения ⋖, ⋗ и ≐ не похожи на обычные арифметические отношения <, >, = ; отношение ≐ не является отношением эквивалентности, а отношения ⋖ и ⋗ не обязательно транзитивны. То есть для отношения предшествования не выполняются некоторые правила, привычные для отношения арифметического порядка. Например, ; ⋗ END но END ⋗ ;

Отношения предшествования удобно занести в матрицу, в которой строки и столбцы помечены терминалами грамматики.

Отношение ≐ означает, что обе лексемы имеют одинаковый уровень предшествования и должны рассматриваться грамматическим процессором в качестве одной конструкции языка. Если для пар отношения предшествования не существует, это означает, что они не могут находиться рядом ни в каком грамматически правильном предложении. Если подобная комбинация лексем встретится в процессе грамматического разбора, то она должна рассматриваться как синтаксическая ошибка.

Существуют алгоритмы автоматического построения матриц предшествования на основе формального описания грамматики. Одна из таких матриц показана на рисунке 3.1.

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 используемой грамматики, однако некоторые из отношений грамматики Паскаля не являются однозначными и метод оперативного предшествования к ним не применим.

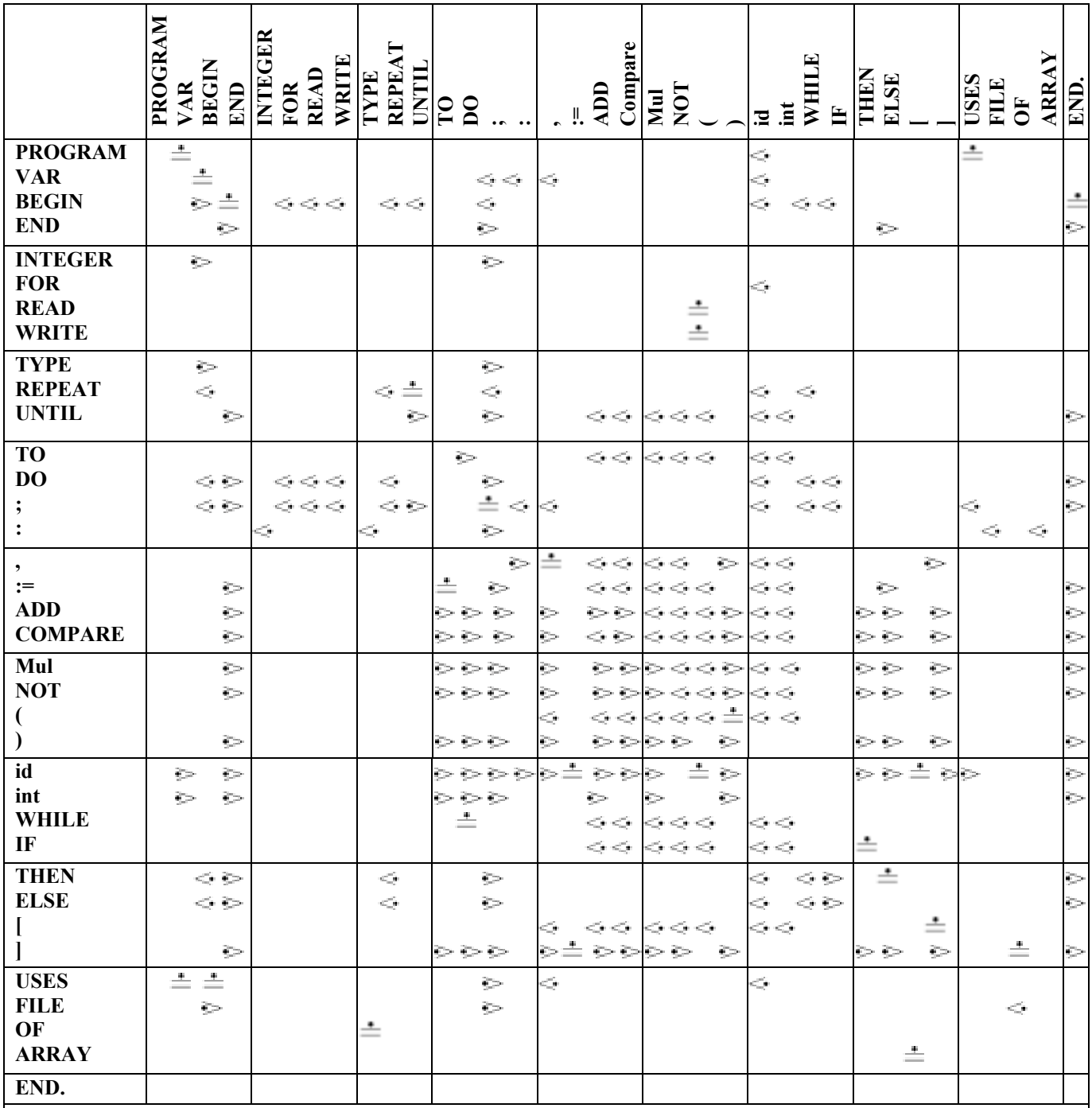
На рисунке 3.2 изображен пошаговый процесс грамматического разбора предложения присваивания в строке 14.

Процесс сканирования слева направо продолжается на каждом шаге грамматического разбора лишь до тех пор, пока не определился очередной фрагмент предложения для грамматического распознавания, то есть первый фрагмент, ограниченный отношениями ⋖ и ⋗. Как только подобный фрагмент

выделен, он интерпретируется как некоторый очередной нетерминальный символ в соответствии с каким-либо правилом грамматики.

Этот процесс продолжается до тех пор, пока предложение не будет распознано целиком. Следует обратить внимание на то, что каждый фрагмент дерева грамматического разбора строится, начиная с оконечных узлов вверх, в сторону корня дерева. Отсюда и возник термин «восходящий разбор».

Для метода операторного предшествования имена нетерминальных символов несущественны. Таким образом, вся информация о грамматике и синтаксических правилах языка содержится в матрице операторного предшествования.



Types (Тип) один из стандартных типов Паскаля (BYTE, INTEGER, SHORTINT, WORD, BOOLEAN, LONGINT, STRING, CHAR, POINTER, REAL, SINGLE, DOUBLE, EXTENDED, COMP, TEXT).

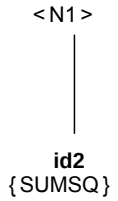
Add (Операция типа Сложение) одна из операций Паскаля равная по приоритету операции сложения (+, -, OR).

Mul (Операция типа Умножение) одна из операций Паскаля равная по приоритету операции умножения (*, /, DIV, MOD, AND, XOR, SHR, SHL).

Compare (Операция типа Сравнение) одна из операций Паскаля равная по приоритету операции сравнения (<>, >=, <=, =, >, <).

Рисунок 3.1 - Матрица операторного предшествования

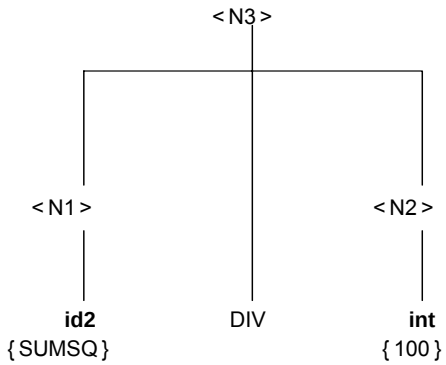
1. ... id1 := id2 DIV
   < = < >



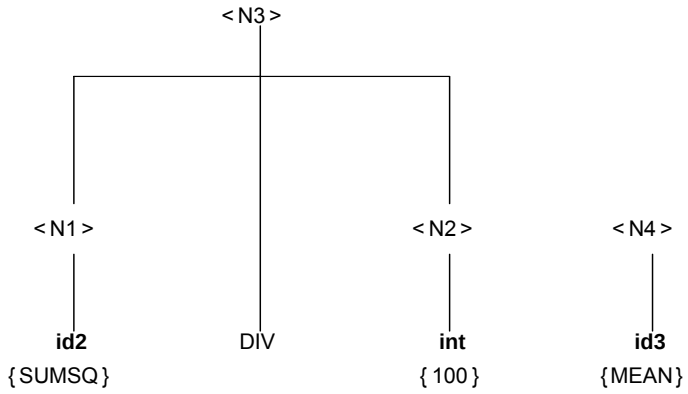
2. ... id1 := < N1 > DIV int -
   < = < < >

3. ... id1 := < N1 > DIV < N2 > -
   < = < >

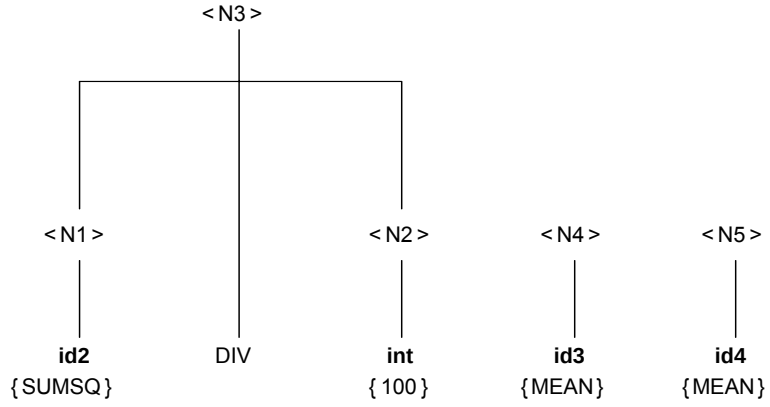
4. ... id1 := < N3 > - id3 *
   < = < < >



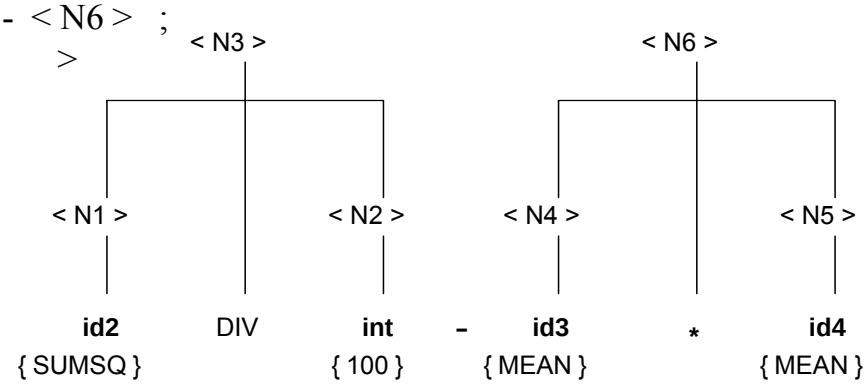
5. ... id1 := < N3 > - < N4 > * id4 ;
   < = < < < >



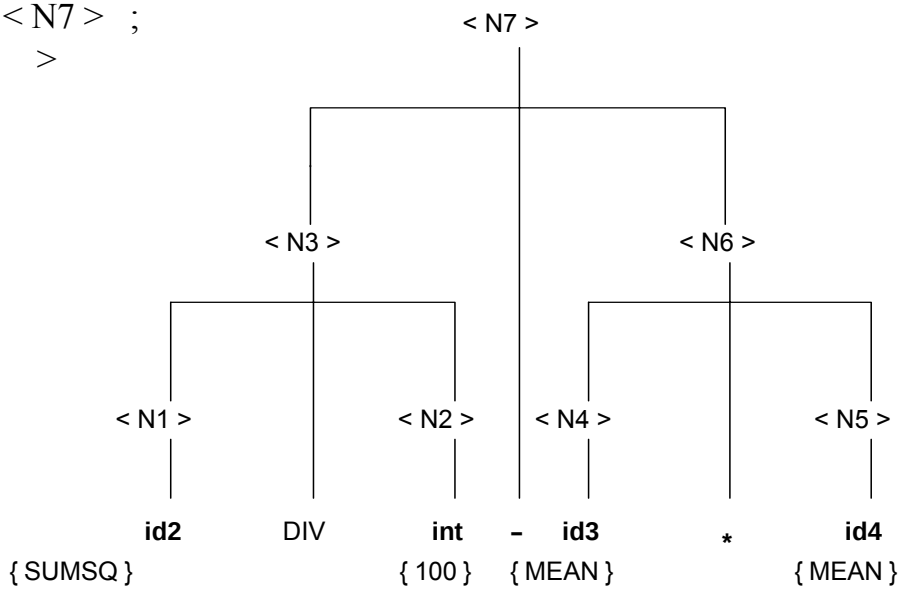
6. ... id1 := < N3 > - < N4 > * <N5> ;
   < = < < >



7. ... id1 := < N3 > - < N6 > ;
< = < >



8. ... id1 := < N7 > ;
< = >



9. ... < N8 > ;

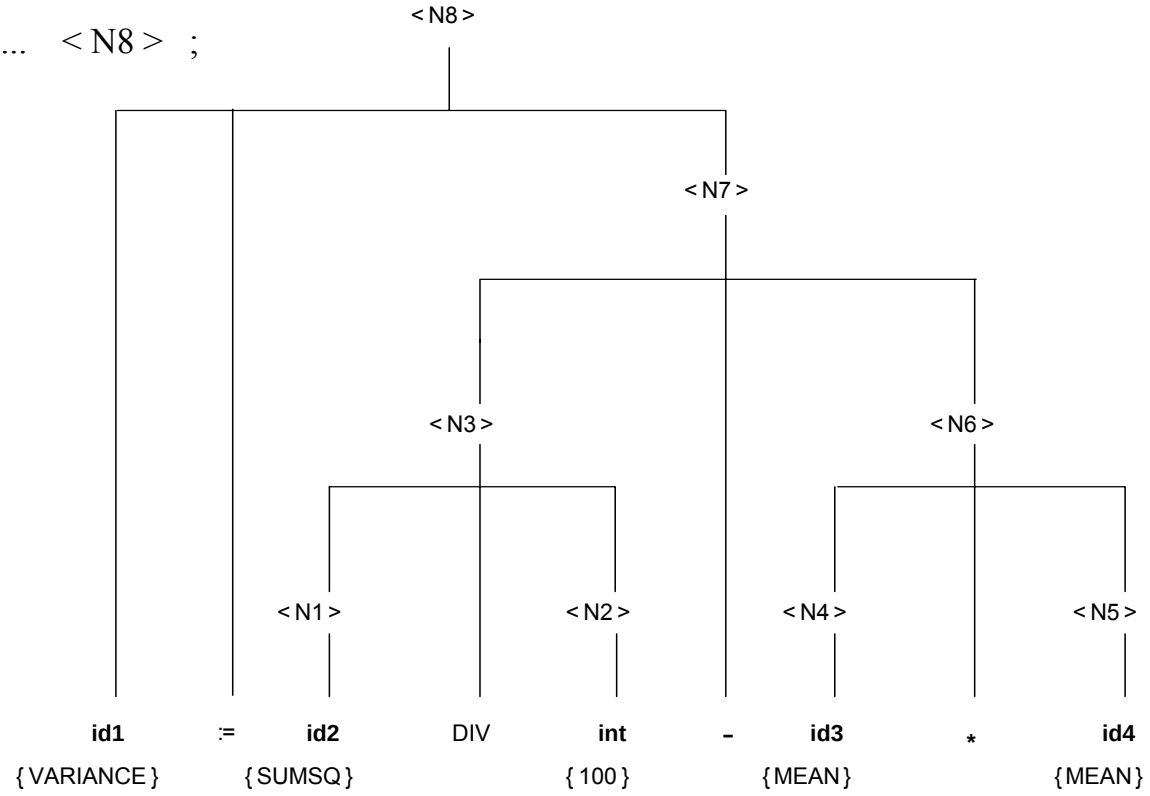
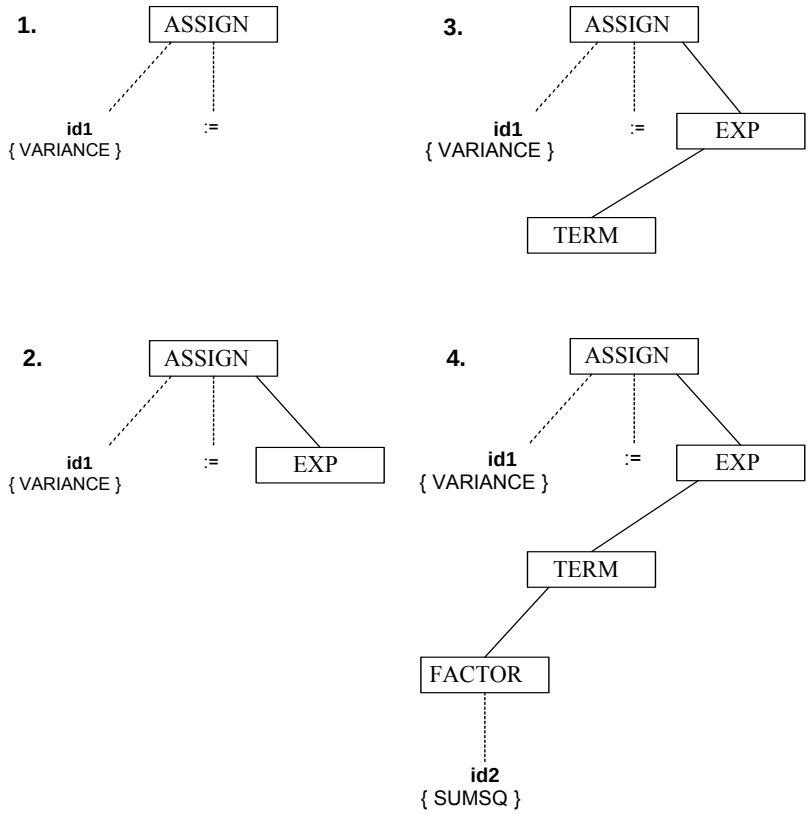


Рисунок 3.2 - Грамматический разбор предложения присваивания методом операторного предшествования

Другой метод грамматического разбора - **нисходящий метод**, называемый рекурсивным спуском. Процессор грамматического разбора, основанный на этом методе, состоит из отдельных процедур для каждого нетерминального символа, определенного в грамматике. Каждая такая процедура ищет во входном потоке подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие подобные процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов. Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает свою работу, передает в вызвавшую ее программу признак успешного завершения и устанавливает указатель текущей лексемы на первую лексему после распознанной подстроки. Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком ошибки или вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

На рисунке 3.3 представлен разбор методом рекурсивного спуска оператора присваивания в строке 14 на Рисунке 1.1.



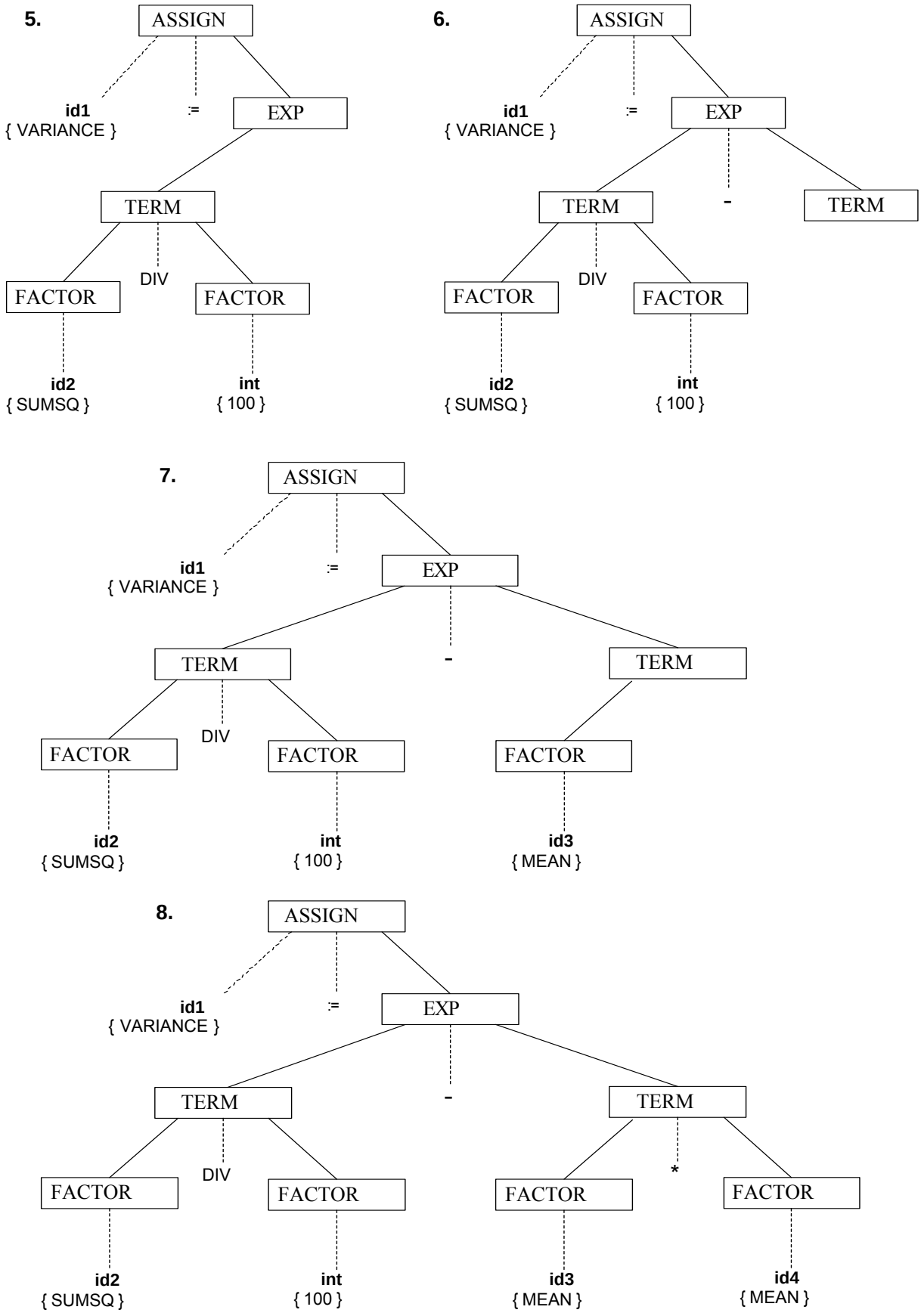


Рисунок 3.3 - Грамматический разбор предложения присваивания методом рекурсивного спуска

4 Генерация кода

Программы генерации кода предназначены для использования с грамматикой на рисунке 1.2. Ни один из методов грамматического разбора не распознает в точности те конструкции, которые описаны грамматикой. Метод операторного предшествования игнорирует некоторые нетерминальные символы, а метод рекурсивного спуска вынужден использовать несколько модифицированную грамматику

Основная работа при грамматическом разборе предложения присваивания состоит в анализе нетерминального символа <exp> в правой части оператора присваивания. В процессе грамматического разбора идентификатор SUMSQ распознается сначала как <factor>, потом как <term>. Далее распознается целое число 100 как <factor>, потом как <term>; затем фрагмент SUMSQ DIV 100 распознается как <term> и т.д. Порядок распознавания фрагментов этого предложения совпадает с порядком, в котором должны выполняться соответствующие вычисления; сначала вычисляются подвыражения SUMSQ DIV 100 и MEAN*MEAN, а затем второй результат вычитается из первого.

Как только очередной фрагмент предложения распознан, вызывается соответствующая программа генерации кода. Например, код, соответствующий правилу $\langle term \rangle_1 := \langle term \rangle_2 * \langle factor \rangle$ будет генерироваться следующим образом.

Программы генерации кода выполняют арифметические операции с использованием регистра AX, и нужно заведомо сгенерировать в объектном коде операцию MUL. Результат этого умножения $\langle term \rangle_1$ после операции MUL сохранится в регистре AX. Если либо $\langle term \rangle_2$, либо <factor> уже находятся в регистре AX(AL) (после выполнения предыдущих операций), нужно только сгенерировать инструкцию MUL. Иначе необходимо сгенерировать также инструкцию MOV, предшествующую инструкции MUL. В этом случае также надо предварительно сохранить значение регистра AX(AL), если это необходимо. Очевидно,

```
MOV     AX,SUMSQ
MOV     DL,100
IDIV    DL
MOV     T1,AL
MOV     AL,MEAN
MOV     DL,MEAN
IMUL    DL
MOV     DL,T1
XOR     DH
SUB     DX,AX
MOV     VARIANCE,DX
```

Рисунок 4.1 — Генерация объектного кода для предложения присваивания

что необходимо отслеживать значение, помещенное в регистр AX(AL), после каждого фрагмента генерируемого объектного кода. Один из вариантов генерации объектного кода приведен на рисунке 4.1.

Процесс генерации кода является машинно-зависимым, поскольку необходимо знать систему команд компьютера, для которого этот код генерируется. Существуют, однако, более сложные проблемы, вытекающие из машинной зависимости, которые возникают при распределении регистров и перестановки машинных инструкций для повышения эффективности. Такого рода оптимизация кода обычно осуществляется с использованием промежуточной формы представления компилируемой программы. В этой промежуточной форме синтаксис и семантика исходных предложений программы уже полностью проанализированы, но трансляция в машинные коды еще не осуществлена. Анализировать и модернизировать программу, представленную в такой промежуточной форме для оптимизации кода, существенно легче, чем для исходной программы на языке высокого уровня или для этой программы, представленной в машинных кодах.

Промежуточные коды можно проектировать на различных уровнях. Иногда промежуточный код получают, просто разбивая сложные структуры исходного языка на более удобные для обращения элементы. Однако чаще в качестве промежуточного кода используют какой-либо обобщенный машинный код. Существует несколько видов таких обобщенных кодов.

Тетрады представляют собой запись операций в форме из четырех составляющих: операция, два операнда и результат операции.

<операция> (<операнд1>,< операнд2>,<результат>)

Тетрады – это линейная последовательность команд, поэтому для них несложно написать алгоритм, который будет преобразовывать последовательность тетрад в последовательность команд результирующей программы либо в последовательность команд Ассемблера. Тетрады не зависят от архитектуры вычислительной системы, на которую ориентированы результирующая программа. Поэтому они представляют собой машинно-независимую форму внутреннего представления программы. Такой код часто называют трехадресным, т.е. два адреса для операндов (кроме случая унарных операций) и один – для результата. Например, для выражения (-a+b)*(c+d) можно представить тетрады следующим образом:

-a = 1
1+b=2
c+d=3
2*3=4

Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда.

<операция> (<операнд1>,< операнд2>)

Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие.

Так же, как и в случае с тетрадами, алгоритм преобразования последовательности триад в команды достаточно прост, но здесь требуется также и алгоритм, отвечающий за распределение памяти, необходимый для хранения промежуточных результатов вычислений, так как временные переменные для этой цели не используются.

Триады требуют меньше памяти для своего представления, чем тетрады, кроме того, они явно отражают взаимосвязь операций между собой, что делает их применение удобным. Триады ближе к двухадресным машинам, чем тетрады. Выражение

a+b+c*d

можно представить в виде тетрад:

a+b=1
c*d=2
1+2=3

Это же выражение можно представить в виде триад:

a+b
c*d
1+2

Видно, что запись в виде триад является более компактной, но, если присутствует фаза оптимизации, их применение затруднительно. Наилучшее решение этой проблемы – косвенные триады, т.е. операнд, ссылавшийся на ранее вычисленную триаду, теперь ссылается на элемент таблицы указателей на триады, а не на саму триаду.

Как триады, так и тетрады можно распространить не только на арифметические выражения, но и на другие конструкции языка. Например, языковую конструкцию if – then – else можно рассматривать как выражение с тремя операндами, которому потребуются четыре адреса как тетраде и три – как триаде.

5 Оптимизация

Оптимизацией называется обработка, связанная с переупорядочиванием и изменением операций в компилируемой программе в целях получения более эффективного объектного кода. Обычно выделяют машинно-зависимую и машинно-независимую оптимизацию. Под машинно-независимой оптимизацией понимается преобразование исходной программы в ее внутреннем представлении, что означает полную независимость от выходного языка, в отличие от машинно-зависимой, выполняемой на уровне объектной программы.

Среди машинно-независимых методов можно выделить самые основные:

- свертка, т.е. выполнение операций, операнды которых известны во время компиляции;
- исключение лишних операций за счет однократного программирования общих подвыражений;
- вынесение из цикла операций, операнды которых не изменяются внутри цикла.

Линейным участком является выполняемая по порядку последовательность операций с одним входом и одним выходом (первая и последняя операции соответственно). Например, последовательность операций представленных ниже образуют линейный участок:

```
I:=1+1;
I:=3;
B:=7+I;
```

Внутри линейного участка обычно проводят две оптимизации: свертку и устранение лишних операций.

Свертка – *это выполнение во время компиляции операций исходной программы, для которых значения операндов уже известны, и поэтому нет нужды их выполнять во время счета.* Например, внутреннее представление в виде триад линейного участка программы, представленного выше, изображено в следующем виде:

(1) + 1,1
(2) := I,(1)
(3) := I,3
(4) + 7,(3)
(5) := B,(4)

Видно, что первую триаду можно вычислить во время компиляции и заменить результирующей константой. Менее очевидно, что четвертую триаду также можно вычислить, так как к моменту ее обработки известно, что I равно 3. Полученный после свертки результат

(1) := I,2
(2) := I,3
(3) := B,10

Свертка, главным образом, применяется к арифметическим операциям, так как они наиболее часто встречаются в исходной программе. Кроме того, она применяется к операторам преобразования типа. Причем проблема упрощается, если эти преобразования заданы явно, а не подразумеваются по умолчанию.

Процесс свертки операторов, имеющих в качестве операндов константы, понятен и сводиться к внутреннему их вычислению. Свертка операторов, значения которых могут быть определены в результате некоторого анализа, несколько сложнее. Обычно свертку осуществляют только в пределах линейного участка при помощи таблицы T, вначале пустой, содержащей пары (K, A), где A – простая переменная для которой известно текущее значение K. Кроме того, каждая свертываемая триада заменяется новой триадой (C, K, 0),

где C (константа) – новый оператор, для которого не надо генерировать команды,

K – результирующее значение свернутой триады.

Алгоритм свертки последовательно просматривает триады линейного участка. Пример работы данного алгоритма приведен в таблице 1.

Таблица 4.1 - Последовательность шагов алгоритма свертки

	Шаг 1	*Шаг 2*	*Шаг 3*	*Шаг 4*	*Шаг 5*
1	+ 1,1	C 2	C 2	C 2	C 2
2	:= I,(1)	:= I,(1)	:= I,2	:= I,2	:= I,(1)
3	:= I,3	:= I,3	:= I,3	:= I,3	:= I,3
4	+ 7,(3)	+ 7,(3)	+ 7,(3)	C 10	C 10
5	:= B,(4)	:= B,(4)	:= B,(4)	:= B,(4)	:= B,10
T			(I,2)	(I,3)	(I,3)
T					(B,10)

С точки зрения работы компилятора, процесс свертки является дополнительным проходом по внутреннему представлению исходной программы, представленной триадами. Но свертку можно проводить и с тетрадами. Кроме того, можно оптимизировать программу в семантических программах во время получения внутреннего представления, и при этом отпадает необходимость в дополнительном проходе. Например, для выражения A:=B+C при восходящем грамматическом разборе программа должна выполнить только одну дополнительную проверку семантик B и C. Если они константы или их значения известны, то программа их складывает и связывает результат с A. В данном случае можно использовать таблицу переменных с известными значениями, которая должна сбрасываться в местах генерации команд передачи управления.

Исключение лишних операций - *i-я операция линейного участка считается лишней, если существует более ранняя идентичная j-я операция и никакая переменная, от которой зависит эта операция, не изменяется третьей операцией лежащей между I-й и j-й операциями.* Например, на линейном участке

D:=D+C*B;

A:=D+C*B;

можно выделить лишние операции. Если представить линейный участок в виде триад, как это представлено ниже, то видно, что операция C*B во второй раз лишняя:

(1) * C,B

(2) + D,(1)

(3) := D,(2)

(4) * C,B

(5) + D,(4)

(6) := A,(5)

Лишней триада (4) является из-за того, что ни C ни B не изменяются после триады (1). В отличии от этого триада (5) лишней не является, так как после первого сложения D с C*B триада (3) изменяет значение D.

Алгоритм исключения лишних операций просматривает операции в порядке их появления. И если i-я триада лишняя, так как уже имеется идентичная ей j-я триада, то она заменяется триадой (SAME, j, 0), где операция SAME ничего не делает и не порождает никаких команд при генерации.

Для слежения за внутренней зависимостью переменных и триад можно поставить им в соответствие числа зависимостей (dependency numbers). При этом используются следующие правила:

- для переменной A число зависимости dep(A) выбирается равным нулю, так как ее значение не зависит ни от одной триады;
- после обработки i-й триады, где переменной A присваивается какое-либо значение число зависимости становиться равным i, так как ее новое значение зависит от i-й триады;
- при обработке i-й триады ее число зависимостей dep(i) становиться равным максимальному из чисел зависимостей ее операндов + 1.

Числа зависимостей используются следующим образом: если i-я триада идентична j-й триаде (j<i), то i-я триада является лишней тогда и только тогда, когда dep(i)=dep(j).

Таблица 4.2 – Пример исключения лишних операций

Обрабатываемая триада I	Dep (переменная) A B C D	Dep(i)	Триада, полученная в результате
(1) * C,B	0 0 0 0	1	(1) * C,B
(2) + D,(1)	0 0 0 0	2	(2) + D,(1)
(3) :=D,(2)	0 0 0 0	3	(3) :=D,(2)
(4) * C,B	0 0 0 3	1	(4) SAME 1
(5) + D,(4)	0 0 0 3	4	(5) + D,(4)
(6) :=A,(5)	0 0 0 3	5	(6) :=A,(5)
	6 0 0 3		

Вынесение инвариантных операций за тело цикла. *Инвариантными называются такие операции, при которых ни один из операндов не изменяется внутри цикла.*

В общем случае данная оптимизация выполняется двойным просмотром тела цикла с анализом операндов каждой из операций внутри цикла. В случае

если они инвариантны, операция выносится назад за тело цикла, а внутри цикла выбрасывается. Для проверки инвариантности переменных используют специальную таблицу инвариантности.

Во время первого прохода цикла заполняется таблица инвариантных переменных, во время второго – выполняется собственно вынесение операций.

6. Выполнение лабораторной работы

Лабораторная работа выполняется с помощью программной модели. Целью данной лабораторной работы является закрепление теоретических знаний о процессе компиляции, полученных при изучении дисциплины "Системное программное обеспечение".

ВНИМАНИЕ!!!

1. *Перед выполнением лабораторной работы необходимо прочитать теорию и пройти тест*.
2. *Перед выполнением какого-либо этапа работы необходимо прочитать особенности программной модели*.
3. *Нельзя работать методом подбора* - все нажатия на кнопки проверки фиксируются и, если задание было выполнено неверно, считаются ошибками и запоминаются.

В работе используется упрощенный синтаксис языка Pascal.

Лабораторный практикум представляет собой программу исследования процесса компиляции, в которой можно выделить четыре основных шага, соответствующих этапам компиляции.

Данные шаги выполняются последовательно и включают в себя следующие задания:

Лексический анализ:

- выделение переменных и констант;
- расстановка приоритетов терминальных символов;
- разбор исходного текста на лексемы.

Синтаксический анализ:

- проведение синтаксического разбора восходящим методом, что включает в себя:

  1) использование программы во внутреннем представлении в виде триад;

  2) демонстрацию возможности использования тетрад;

- проведение синтаксического разбора нисходящим методом – построение дерева синтаксического разбора.

Оптимизация:

- проведение оптимизации методом свёртки;
- проведение оптимизации методом исключения лишних операций.

Генерация кода:

- генерация выходного кода на языке Ассемблера на основе построенной на предыдущем этапе таблицы триад.

Интерфейс программной модели показан на рисунке 6.1. Каждая вкладка соответствует одному из шагов компиляции. Вкладка становится доступной только после правильного выполнения предыдущего шага компиляции.

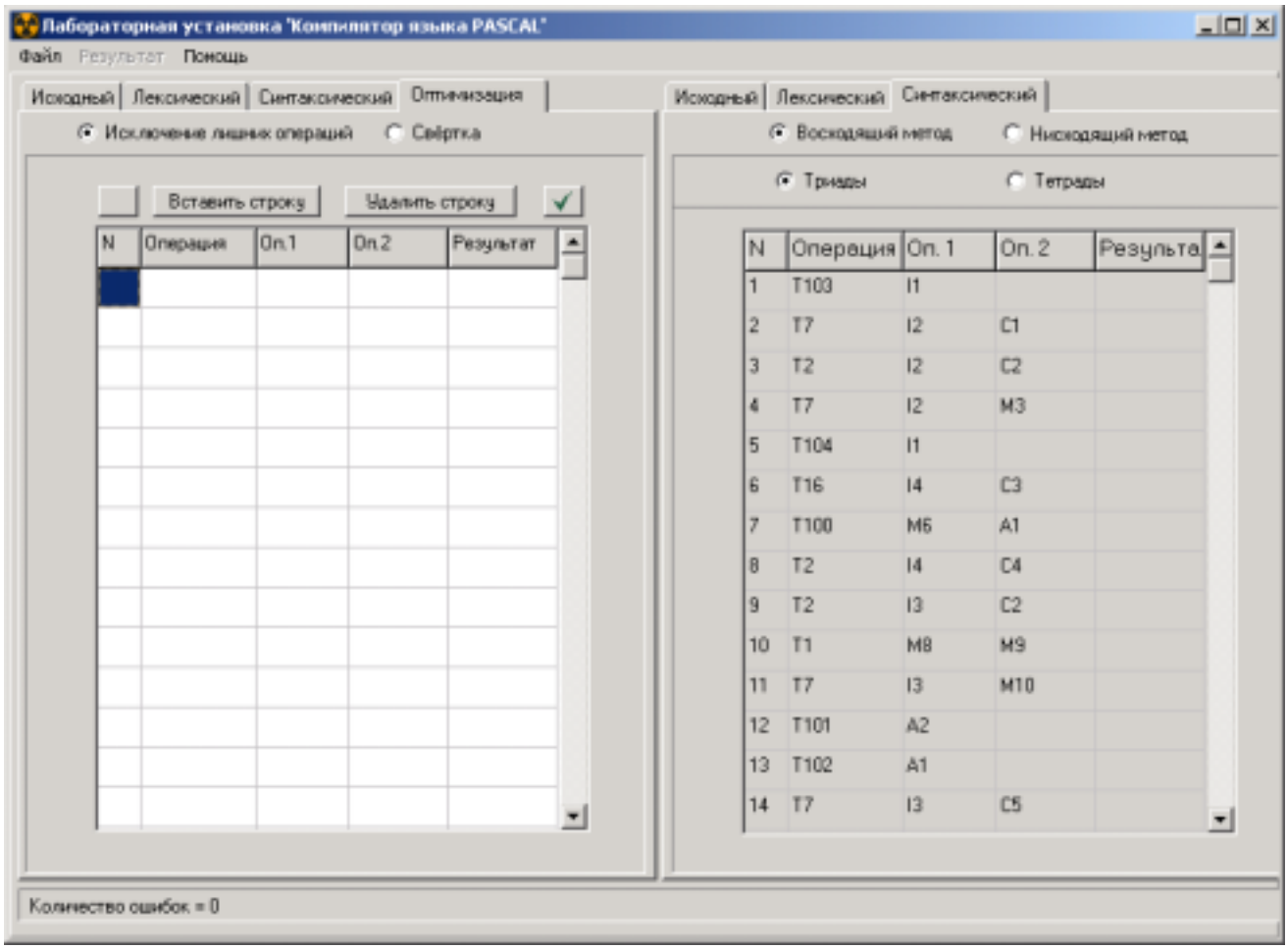


Рисунок 6.1 – Интерфейс программной модели

Выполнение лабораторной работы начинается с прохождения теста. **Студенты, не прошедшие тест, к лабораторной работе не допускаются.**

После прохождения теста загружается исходный текст программы на языке Pascal. Номер варианта лабораторной работы соответствует номеру компьютера, за которым выполняется работа.

После загрузки программы доступна вкладка «Лексический анализ».

На вкладке «Лексический анализ» расположены три внутренние служебные таблицы, которые предлагается заполнить.

После окончания работы с каждой из таблиц необходимо проверить правильность ее заполнения. Для этого необходимо нажать на кнопку "*Проверить*" (кнопка с зелёной галочкой), расположенную над соответствующей таблицей. При неверном заполнении какого-либо из элементов таблицы будет выдано соответствующее сообщение с указанием места ошибки. Если таблица заполнена правильно, то произойдет изменение ее фона на более темный и кнопка "*Проверить*" данной таблицы станет неактивной.

Таблица терминальных символов

В данной таблице поля "*символ*" и "*индекс*" заполнены программно и являются неизменными для всех вариантов заданий. Необходимо только правильно расставить приоритеты между операциями. При этом необходимо учитывать следующие особенности данной версии лабораторной установки:

- терминалы *begin*, *end*, *end.*, ***;*** и *else* имеют один и тот же приоритет;
- приоритет операций сравнения и закрывающей скобки один и тот же;
- приоритеты начинаются с нуля (низший приоритет) и заканчиваются шестью (высший приоритет).

Для расстановки приоритетов удобно воспользоваться следующей таблицей:

Приоритет	
0	**Begin**, **End**, **End.** , **;**, **Else**
1	**Then** , **Call**
2	**:=** , **If**
3	= , > , < , **)**
4	+ ,
5	* , /
6	**(**

Таблица констант

Необходимо заполнить все поля. При этом важно учитывать:

- индексы начинаются с единицы;
- константы заносятся по мере их появления в исходной программе.

Таблица переменных

Необходимо заполнить все поля. При этом должны соблюдаться следующие правила:

- индексы начинаются с единицы;
- первыми заносятся переменные процедур;
- поле "*место*" определяет местонахождение переменной

0 - основная программа;

1, 2, 3... - процедуры по порядку их появления в теле программы;

- в поле "*тип*" для заголовка процедуры вводиться *procname.*

После заполнения всех таблиц автоматически произойдет переход к Таблице лексем.

Таблица лексем

В данной таблице поле " *символ*" заполнять не обязательно - это поле является вспомогательным, введенным для удобства работы на следующих этапах. Поля "тип" и "индекс" выбираются из заполненных ранее внутренних таблиц. При этом для поля "тип" существуют следующие варианты заполнения:

T - терминальный символ;

I - переменная;

C - константа.

Заполнение таблицы идет последовательно по тексту программы.

При заполнении необходимо использовать терминал

T103 - начало процедуры с символом PROCEDURE.

После правильного заполнения таблицы станет доступной вкладка «Синтаксический анализ».

Вкладка «Синтаксический анализ» предназначена для работы с таблицей триад и тетрад.

Таблица триад

Общие принципы работы с таблицей триад такие же как у таблицы лексем. При этом есть некоторые особенности, которые необходимо учитывать:

- не нужно заполнять поле *"результат"*, так как оно используется только в случае применения тетрад, как формы внутреннего представления программы;
  Формат полей "операция", "операнд 1", "операнд 2" имеет вид ***ТИ***,
  где ***И*** - индекс символа в соответствующей таблице или порядковый номер, а ***Т*** - символ– идентификатор, принимающий следующие значения:
  M - строка матрицы;
  T - таблица терминалов;
  C - таблица констант;
  I - таблица переменных;
  A - метка перехода.
  Например, запись T1 означает что используется терминальный символ имеющий в таблице терминалов индекс 1 (т.е. операция сложения);
- кроме терминальных символов из таблицы терминалов, представленной ранее, используются дополнительные терминальные символы, такие, как:
  T100 - JF – Переход при невыполнении условия;
  T101 - JMP – Безусловный переход;
  T102 - Mark – Метка;
  T103 - Procedure – Начало процедуры;
  T104 - Procend – Конец процедуры;
- в случае отсутствия операнда поле соответствующего операнда остается незаполненным;
- нет необходимости обозначать конец и начало основной программы.

После правильного заполнения таблицы триад проводится демонстрация вида данного внутреннего представления для случая тетрад. При этом вводится новый символ **R,** обозначающий переменную памяти. Данный символ применяется вместо символа **M**. Кроме того, заполняется поле "*результат* ".

После окончания работы с таблицей триад и демонстрации таблицы тетрад необходимо переключиться на закладку «Нисходящий метод».

Вкладка «Нисходящий метод» предназначена для работы с деревом синтаксического разбора.

Дерево строится по основной программе; для процедур строятся только их вызовы.

Построение дерева

Главная ветвь дерева называется *idProgram* и изначально находится на форме построения дерева. *Все остальные ветви являются ее подветвями !!!*

Следующим уровнем являются секции: секция операций и секция переменных, причем секция переменных должна быть первой. В ней последовательно заносятся переменные и их типы.

Помните, что секция программ начинается с *Begin* и заканчивается *End.*

Ввод новых ветвей и редактирование старых производится нажатием на правую кнопку мыши на форме построения дерева, при этом появляется меню выбора операций.

При создании новой ветви дерева появляется форма "Добавление элемента". При этом в поле "*идентификатор*" необходимо правильно выбрать тип идентификатора. Существуют следующие типы:

idVariableSection - Секция переменных;

idOperation - SectionСекция операций;

idOperator - Оператор, т.е. строка заканчивающаяся точкой с запятой;

idExpression - Выражение, т.е. часть оператора, заключенная в скобки (сами скобки не относятся к выражению);

idKeyWord - Ключевое слово (например, VAR);

idIdent - Имя переменной;

idConst - Имя константы;

idSymbol - Символ (знаки операций, точка с запятой и т.д.);

idType - Тип переменной.

После построения правильного дерева синтаксического разбора становится доступной вкладка «Оптимизация».

Вкладка «Оптимизация» предназначена для оптимизации объектного модуля двумя методами. На этапе «Исключение лишних операций» рекомендуется четко следовать алгоритму, описанному в теории, даже если решение кажется очевидным, так как не все одинаковые операции можно исключать на данном этапе. В таблицу необходимо вносить только окончательный вариант работы алгоритма.

После правильного заполнения таблицы необходимо перейти к этапу «Свертка».

На этапе «Свертка» предлагается повторно заполнить таблицу триад, но с оптимизированными триадами

Для ввода добавочных констант нужно воспользоваться кнопкой расположенной внизу таблицы.

После правильного заполнения данной таблицы необходимо перейти к этапу «Генерация»

Вкладка «Генерация» предназначена для построения выходного кода на языке Ассемблера.

Внимание!!! Выходной код строиться по строгим правилам перевода триад в ассемблерный код. При написании вместо пробелов использовать символы табуляции.

В общем случае структура на языке Ассемблера должна быть следующего вида:

```
.MODEL small
.DATA
.....
.CODE
PROC
.....
ENDP
Main:
.....
END Main
```

Секция переменных

- Переменные должны быть описаны в порядке их нахождения в таблице переменных, причем переменные процедур после имени должны иметь подчеркивание и индекс процедуры (Например, если в процедуре с индексом 2 есть переменная MMM, то она должна быть описана MMM_2).
- После описания переменных из таблицы переменных описываются дополнительные переменные, полученные на этапе построения таблицы тетрад (код строиться по ней).
- После переменных описываются константы из таблицы констант.
- Тип всех переменных и констант **dw** (для простоты).

Секция кода

Так как код является не полностью оптимизированным (в программе реализованы только два метода оптимизации), то любая триада переводиться в две, либо три строки ассемблерного кода. Например, триада присваивания сводиться к занесению в регистр AX второго операнда и занесению из регистра AX в ячейку, описанную в поле «результат» таблицы триад.

Оформление отчета

Отчет должен содержать дерево грамматического разбора, построенное восходящим методом, для фрагмента программы, указанного преподавателем.

Библиографический список

1. Гордеев А.В. Системное программное обеспечение /А.В. Гордеев, А.Ю.Молчанов. – СПб.:Питер, 2002. –736с.:ил.
2. Молчанов А.Ю. Системное программное обеспечение: Учеб./ А.Ю.Молчанов. – СПб.:Питер, 2003. —396с.:ил.
3. Пустоваров В.И. Язык Ассемблера в программировании информационных и управляющих систем /Пустоваров В.И. –Киев: М.: Век: Энтроп:Бином универсал, 1996. –304с.
4. Бек Л. Введение в системное программирование /Пер. с англ. /Л.Бек. – М.:Мир, 1988.–448с.
5. Грис Д. Компиляторы для цифровых вычислительных машин: Пер. с англ./ Д.Грис.-М.:Мир, 1989.-486с.
6. Григорьев В. Л. Микропроцессор i486. Архитектура и программирование: В 2 ч. / В. Л. Григорьев. – М.: ГРАНАЛ, 1993.
7. Абель П. Язык Ассемблера для IBM PC и программирования: Пер. с англ. / П. Абель. – М.: Высш. шк., 1992.
8. Фролов А.В. Аппаратное обеспечение IBM PC: В 2 ч. / А.В.Фролов, Г.В.Фролов.– М.: Диалог-МИФИ, 1993.