2017

Глава 4 *Основы конструирования компиляторов*

МГТУ им. Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети
Лектор: д.т.н., проф.
Иванова Галина Сергеевна

Введение. Метод Рутисхаузера

0. Записать в полной скобочной форме:

   $d = a+b*c \Rightarrow d = a+(b*c)$

1. Сопоставить индексы:

```
N[0]:=0
J:=1
Цикл-пока   S[J]≠'_'
    Если  S[J] = '(' или S[J] = <операнд>
          то  N[J]:=N[J-1] +1
          иначе  N[J]:=N[J-1] -1
    Все-если
    J:=J+1
Все-цикл
N[J]:=0
```

2. Определить max индекса $k(k-1)k$ и построить тройку.

3. Удалить обработанные символы из выражения, результату сопоставить индекс $N=k-1$

Пример использования метода Рутисхаузера для разбора выражения

Пример. ((a+b)*c+d)/k ⇒ (((a+b)*c)+d)/k

a) **S: (((a + b) * c) + d) / k**

N: 0 1 2 3 4 3 4 3 2 3 2 1 2 1 0 1 0

=> T1 = a+b

b) **S: ((T1 * c) + d) / k**

N: 0 1 2 3 2 3 2 1 2 1 0 1 0

=> T2 = T1*c

c) **S: (T2 + d) / k**

N: 0 1 2 1 2 1 0 1 0

=> T3 = T2+d

d) **S: T3 / k**

N: 0 1 0 1 0

=> T4 = T3/k

4.1 Основные понятия

4.1.1 *Классификация компилирующих программ*

Транслятор – программа, которая переводит программу, написанную на одном языке, в эквивалентную ей программу, написанную на другом языке.

Компилятор – транслятор с языка высокого уровня на машинный язык или язык ассемблера.

Ассемблер – транслятор с языка Ассемблера на машинный язык.

Интерпретатор – программа, которая принимает исходную программу и выполняет ее, не создавая программы на другом языке.

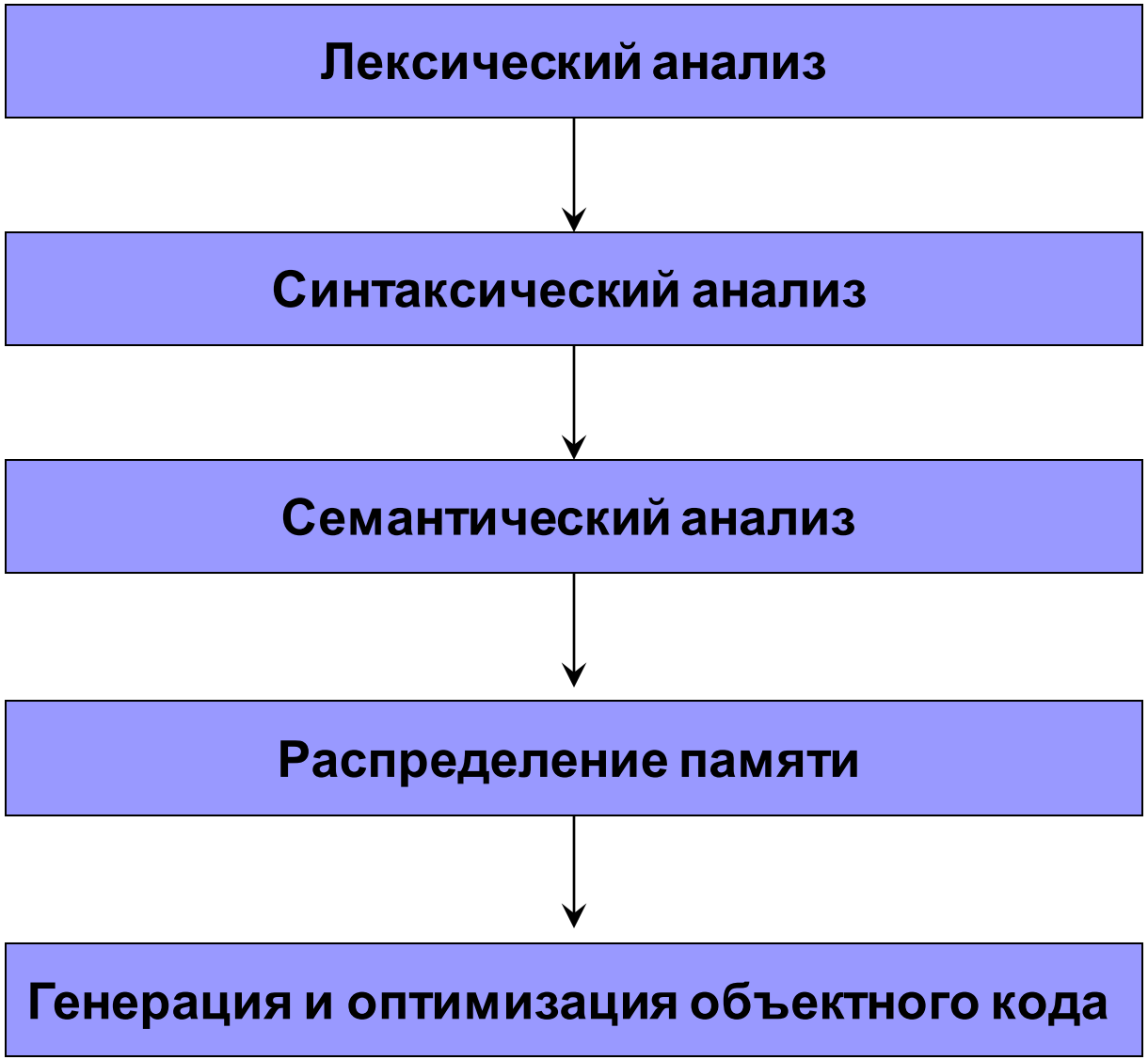
Макрогенератор (для компиляторов – ***препроцессор***) – программа, которая обрабатывает исходную программу, как текст, и выполняет в нем замены указанных символов на подстроки. Макрогенератор обрабатывает программу до трансляции.

4.1.2 *Структура компилирующей программы*

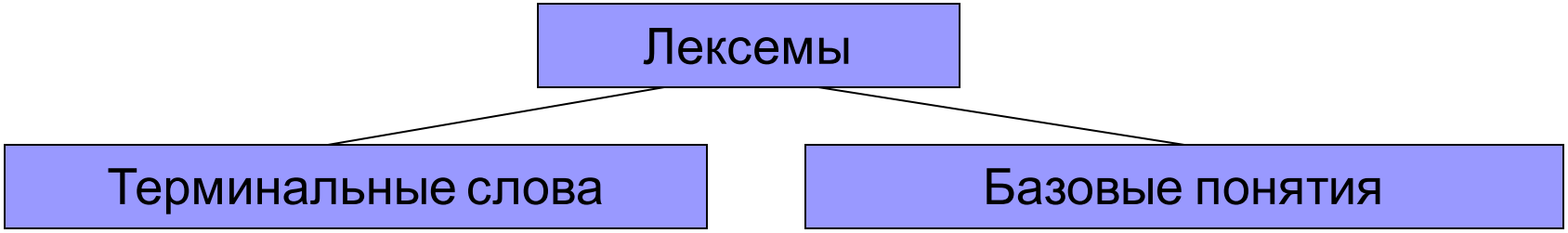
Синтаксис – это совокупность правил, определяющих допустимые конструкции языка, т. е. его *форму*.

Семантика – это совокупность правил, определяющих логическое соответствие между элементами и значением синтаксически корректных предложений, т. е. *содержание* языка.

Структура процесса компиляции



Лексический анализ



Пример. **if Sum>5 then pr:= true;**

Лексема	Тип	Значение	Ссылка
if	Служебное слово	Код «if»	-
Sum	Идентификатор		Адрес в таблице идентиф.
>	Служебный символ	Код «>»	-
5	Литерал		Адрес в таблице литералов
then	Служебное слово	Код «then»	-
pr	Идентификатор		Адрес в таблице идентиф.
:=	Служебный символ	Код «:=»	-
true	Литерал		Адрес в таблице литералов
;	Служебный символ	Код «;»	-

Синтаксический анализ

Таблица токенов

Лексема	Тип	Значение	Ссылка
if	Сс	Код if	
Sum	Ид		Адрес
>	С	Код >	
5	Кц		Адрес
then	Сс	Код then	
pr	Ид		Адрес
:=	С	Код :=	
true	Кл		Адрес
;	Р		

Логическое выражение

Оператор

Условный оператор

4.2 Формальные грамматики и распознаватели

4.2.1 *Формальный язык и формальная грамматика*

Алфавит – непустое конечное множество символов, используемых для записи предложений языка.

Пример:

A = {0,1,2,3,4,5,6,7,8,9,+,-}

Строка – любая последовательность символов алфавита.

A* - множество строк, включая пустую, составленных из A.

A^+ - множество строк за исключением пустой, составленных из A.

$$A^* = A^+ \cup e$$

Формальным языком L в алфавите A называют произвольное подмножество множества *A**.

Язык можно задать *перечислением* и *правилами продукции*.

Формальная грамматика

$$G = (V_T, V_N, P, S),$$

где V_T – алфавит языка или множество терминальных (окончательных, незаменяемых) символов;

V_N – множество нетерминальных (заменяемых) символов – вспомогательный алфавит, символы которого обозначают допустимые *понятия* языка,

$V_T \cap V_N = \varnothing$;

$V = V_T \cup V_N$ – словарь грамматики;

P – множество порождающих правил – каждое правило состоит из пары строк (α, β), где $\alpha \in V+$ – левая часть правила, $\beta \in V^*$ – правая часть правила: $\alpha \rightarrow \beta$, где строка α должна содержать хотя бы один нетерминал;

$S \in V_N$ – начальный символ – аксиома грамматики.

Грамматика записи десятичных чисел G_0

V_T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -};
V_N = {<целое>, <целое без знака>, <цифра>, <знак>};
P = {<целое> → <знак><целое без знака>,
<целое> → <целое без знака>,
<целое без знака> → <цифра><целое без знака>,
<целое без знака> → <цифра>,
<цифра> → 0,
<цифра> → 1,
<цифра> → 2,
<цифра> → 3,
<цифра> → 4,
<цифра> → 5,
<цифра> → 6,
<цифра> → 7,
<цифра> → 8,
<цифра> → 9,
<знак> → +,
<знак> → - };
S = <целое>.

Правосторонняя рекурсия

Форма Бэкуса-Наура (БНФ)

Условные обозначения:

<Имя> – нетерминальный символ – конструкция;

Имя – терминальный символ – символ алфавита;

::= – «можно заменить на»;

| – «или»

Пример:

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

<целое без знака> ::= <цифра><целое без знака>|<цифра>

<цифра > ::= 0|1|2|3|4|5|6|7|8|9

<знак> ::= +| -

4.2.2 *Грамматический разбор*

Выводом называется последовательность подстановок.

Пример. Вывод строки «-45»:

<целое> $\Rightarrow_1$

$\Rightarrow_1$**<знак><целое без знака>** $\Rightarrow_2$

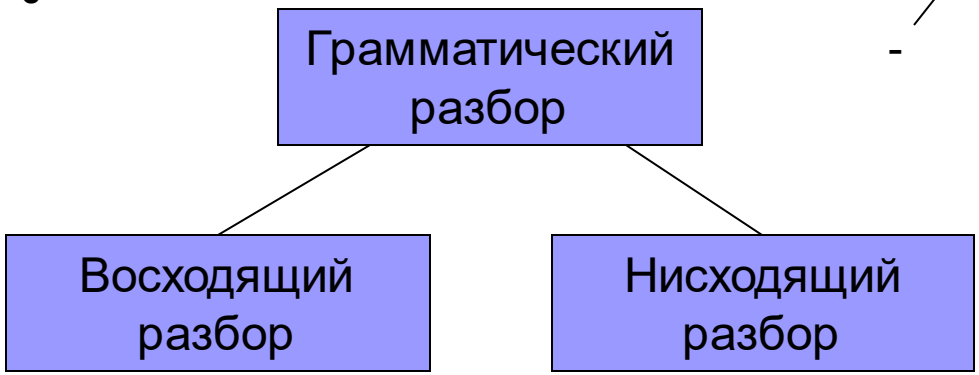
$\Rightarrow_2$ **<знак><цифра><целое без знака>** $\Rightarrow_3$

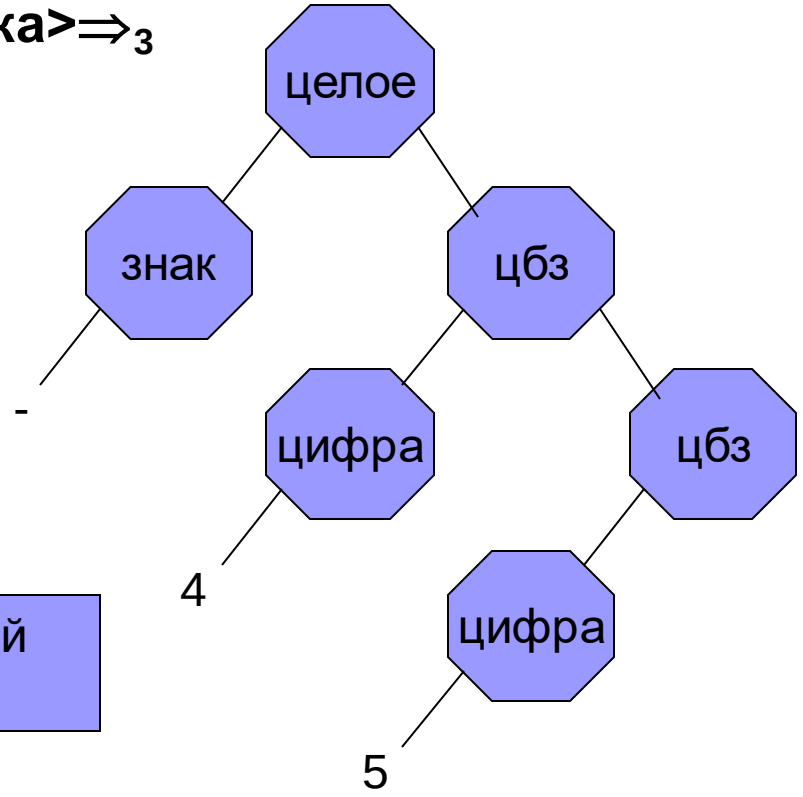
$\Rightarrow_3$ **<знак><цифра><цифра>** $\Rightarrow_4$

$\Rightarrow_4$ **- <цифра><цифра>** $\Rightarrow_5$

$\Rightarrow_5$ **- 4<цифра>** $\Rightarrow_6$

$\Rightarrow_6$ **- 45**





1. Левосторонний нисходящий грамматический разбор

Пример. Разобрать строку «-45» по правилам грамматики:

1) **<целое> ::= <знак><целое без знака>|<целое без знака>**
2) **<целое без знака> ::= <цифра><целое без знака>|<цифра>,**
3) **<цифра > ::= 0|1|2|3|4|5|6|7|8|9,**
4) **<знак> ::= +| - .**

Правосторонняя рекурсия

Альтернативные правила нумеруем буквами а, б, в…

Идея разбора:

Если первый символ правила – терминальный, т.е. символ алфавита, и он совпадает с первым символом распознаваемой строки, то символ считается распознанным и удаляется из стека и строки. Если терминалы не совпадают, то ищем альтернативу ближайшему правилу и производим его замену.

Если первый символ правила в стеке – нетерминал, то заменяем его на первое из правил, его определяющих.

И так далее…

Таблица грамматического разбора

Распознано	Строка	Правила	Правило
	–45◄	<Целое> ◄	?
	–45◄	<Знак><ЦБЗ>◄	1а – ?
	–45◄	+<ЦБЗ>◄ –<ЦБЗ>	4а – нет 4б – да
–	45◄	<ЦБЗ>◄	
–	45◄	<Цифра><ЦБЗ>◄	2а – ?
–	45◄	0..3<ЦБЗ>◄ 4<ЦБЗ>◄	3а..3г – нет 3е – да
–4	5◄	<ЦБЗ>◄	
–4	5◄	<Цифра><ЦБЗ>◄	2а – ?
–4	5◄	0..4<ЦБЗ>◄ 5<ЦБЗ>◄	3а..3е – нет 3ж – да
–45	◄	<ЦБЗ>◄	
–45	◄	<Цифра><ЦБЗ>◄	2а –?
–45	◄	0..9<ЦБЗ>◄	3а..3и – нет
	◄	<ЦБЗ>◄	2б –?
	◄	<Цифра>◄	3а..3и – нет
–4	5◄	<Цифра>◄	2б – ?
	5◄	1..4<ЦБЗ>◄ 5<ЦБЗ>◄	3а..3е – нет 3ж – да
–45	◄	◄	Конец

Черными стрелками показаны смены правил на альтернативные

Красные стрелки показывают возвраты, требующие сменить правило предыдущих уровней, что требует сохранения истории разбора

Возвраты вызваны неправильным выбором альтернатив правил

2. Левосторонний восходящий грамматический разбор

Пример. Разобрать строку «-45», используя правила:

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

<целое без знака> ::= <целое без знака><цифра>|<цифра>,

<цифра > ::= 0|1|2|3|4|5|6|7|8|9,

<знак> ::= +| - .

Левосторонняя рекурсия

Идея метода:

При разборе запись просматривается слева направо и ищется часть, совпадающая с правой частью правила, – основа.

Основа – последовательность символов, сворачиваемая на следующем шаге разбора.

Найденная основа заменяется левой частью соответствующего правила и т.д.

Последовательность выбора основ определена алгоритмом. Неверный выбор основы приводит к необходимости возврата.

2. Левосторонний восходящий грамматический разбор

Последовательность получения сентенциальных форм при разборе:

-45

<знак> 45

<знак> <цифра>5

<знак> <цбз>5

<целое> 5 — Тупик!

<знак> <цбз>5

<знак><целое> 5 — Тупик!

<знак> <цбз>5

<знак> <цбз><цифра>

<целое> <цифра> — Тупик!

<знак> <цбз><цифра>

<знак> <целое> <цифра> — Тупик!

<знак> <цбз><цифра>

<знак> <цбз>

<целое>

<целое> ::= <знак><ЦБЗ>|< ЦБЗ >,
<ЦБЗ> ::= <ЦБЗ><цифра>|<цифра>,
<цифра > ::= 0|1|2|3|4|5|6|7|8|9,
<знак> ::= +| -

Возвраты возникают из-за неверного выбора основы

4.2.3 *Классификация грамматик Хомского*

Тип 0 – грамматики фразовой структуры или грамматики «без ограничений»:

$\alpha \to \beta$, где $\alpha \in V+$, $\beta \in V^*$ – в таких грамматиках допустимо наличие любых правил вывода, что свойственно грамматикам естественных языков;

Тип 1 – контекстно-зависимые (неукорачивающие) грамматики:

$\alpha \to \beta$, где $\alpha \in V^+$, $\beta \in V^*$, $|\alpha| \leq |\beta|$ – в этих грамматиках для правил вида $\alpha \boldsymbol{X} \beta \to \alpha \boldsymbol{x} \beta$ возможность подстановки строки *x* вместо символа *X* определяется присутствием подстрок α и β, т. е. *контекста*, что также свойственно грамматикам естественных языков;

Тип 2 – контекстно-свободные грамматики:

$A \to \beta$, где $A \in V_N$, $\beta \in V^*$ – поскольку в левой части правила стоит нетерминал, подстановки не зависят от контекста;

Тип 3 – регулярные грамматики:

$A \to \alpha$, $A \to \alpha B$, где A, $B \in V_N$, $\alpha \in V_T$.

4.2.4 *Распознавание регулярных грамматик*

4.2.4.1 Конечный автомат

$$M = (Q, \Sigma, \delta, q_0, F),$$

где Q – конечное множество состояний;

Σ – конечное множество входных символов;

$\delta(q_i, c_k)$ – функция переходов (q_i – текущее состояние, c_k – очередной символ);

q_0 – начальное состояние;

$F = \{q_j\}$ – подмножество допускающих состояний.

Пример. Автомат «Чет-нечет»:

Q = {Чет, Нечет};

Σ = {0, 1};

δ(Чет, 0) = Чет, δ(Нечет, 0) = Нечет,

δ(Чет, 1) = Нечет, δ(Нечет, 1) = Чет;

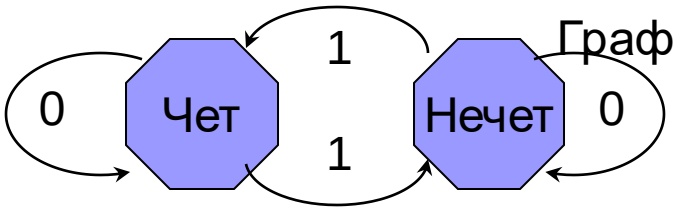
q_0 = Чет;

F = {Чет}

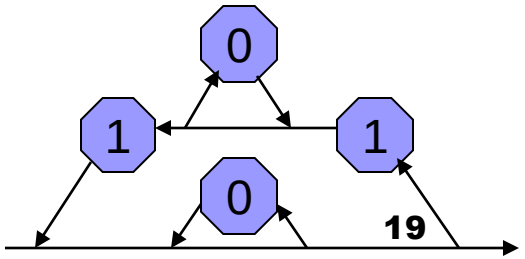
Таблица переходов

	0	1
Чет	Чет	Нечет
Нечет	Нечет	Чет

Граф переходов



Синтаксическая диаграмма



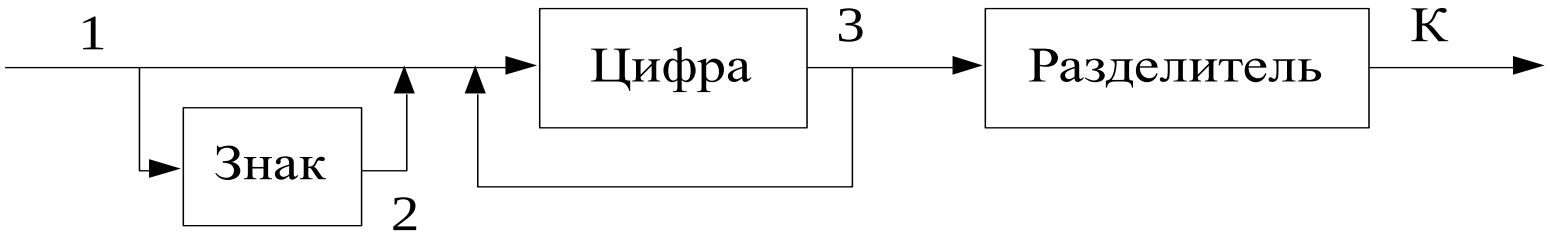
Программная реализация конечного автомата

	0	1	Символы заверш.	Другие символы
Чет	Чет	Нечет	Конец	Ошибка
Нечет	Нечет	Чет	Ошибка	Ошибка

Ind := 1
_q_ := _q_0
Цикл-пока _q_ ≠ «Ошибка» и _q_ ≠ «Конец»
    _q_ = _Table_ [_q_, S[_Ind_]]
    Ind := _Ind_ +1
Все-цикл
Если _q_ = «Конец»
    то «Строка принята»
    иначе «Строка отвергнута»
Все-если

4.2.4.2 Лексические анализаторы

Пример. Распознаватель целых чисел.



Состо-яние	Знак	Циф-ра	Разде-литель	Другие
1	2, A1	3, A2	E, D1	E, D4
2	E, D2	3, A2	E, D3	E, D4
3	K, A3	3, A2	K, A3	E, D4

A0: Инициализация:
Целое := 0;
Знак_числа := «+».
A1: Знак_числа := Знак
A2: Целое := Целое*10 + Цифра
A3: Если Знак_числа = «-» то
Целое := -Целое
Все-если

Д1: «Строка не является числом»;
Д2: «Два знака рядом»;
Д3: «В строке отсутствуют цифры»,
Д4: «В строке встречаются недопустимые символы»

Распознаватель целых чисел

```
Ind := 1
q := 1
Выполнить A0
Цикл-пока q ≠ «E» и q ≠ «K»
    Если S[Ind] = «+» или S[Ind] = «-»,
          то j := 1
    иначе
          Если S[Ind] ≥ «0» и S[Ind] ≤ «9»,
                то    j := 2,
                 иначе  Если  S[Ind] э РАЗДЕЛИТЕЛИ
                                  то j := 3
                                  иначе   j := 4
                            Все-если
          Все-если
    Все-если
    Выполнить Ai := Table [q, j]. A()
    q := Table [q, j]
    Ind := Ind +1
Все-цикл
```

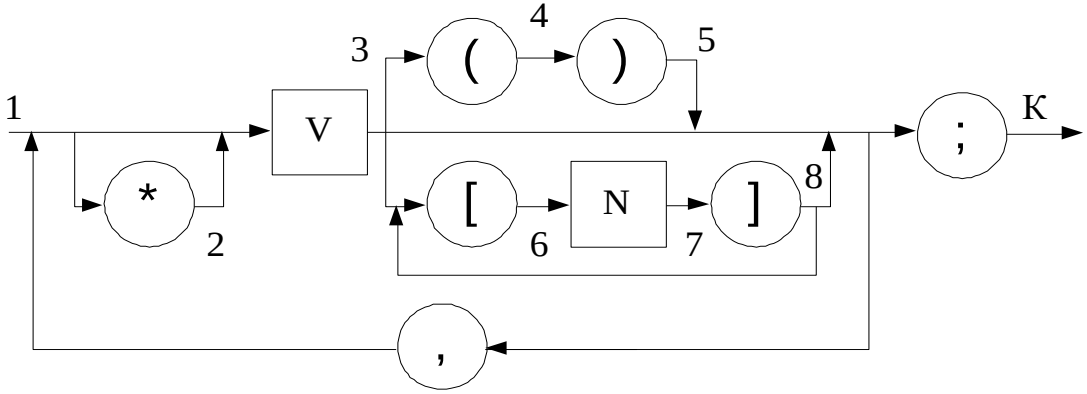
```
Если q = «K»
     то  Выполнить A3
          Вывести «Это число»
     иначе Вывести сообщение Di
Все-если
```

Состоя-ние	Знак	Циф-ра	Разде-литель	Дру-гие
1	2, A1	3, A2	E, D1	E, D4
2	E, D2	3, A2	E, D3	E, D4
3	K, A3	3, A2	K, A3	E, D4

4.2.4.3 Синтаксические анализаторы

Пример. Синтаксический анализатор списка описания целых скаляров, массивов и функций, например:

```
int xaf, y22[5], zrr[2][4], re[N], fun(), *g;
int V,V[N],V[N][N],V[N],V(),*V;
```



Обозначения:
V – идентификатор;
N – целочисленная константа;
служебные символы: «[ ] ( ) , ; *».

	V	N	*	(	)	[	]	,	;	Другие
1	**3**	E	**2**	E	E	E	E	E	E	E
2	**3**	E	E	E	E	E	E	E	E	E
3	E	E	E	**4**	E	**6**	E	**1**	**К**	E
4	E	E	E	E	**5**	E	E	E	E	E
5	E	E	E	E	E	E	E	**1**	**К**	E
6	E	**7**	E	E	E	E	E	E	E	E
7	E	E	E	E	E	E	**8**	E	E	E
8	E	E	E	E	E	**6**	E	**1**	**К**	E

Алгоритм анализатора

Ind := 1

q := 1

Цикл-пока *q* ≠ «E» и *q* ≠ «K»

    q := *Table* [*q*, *Pos*(*S*[*Ind*], «VN*()[];»)]

    Ind := *Ind* +1

Все-цикл

Если *q* = «K»

    то Вывести сообщение «Да»

    иначе Вывести сообщение «Нет»

Все-если

где *Pos*(*S*[*Ind*], «VN*()[];») – позиция (номер столбца таблицы) очередного символа в строке символов или 0, если это «другой» символ

4.2.5 *Распознавание КС-грамматик*

4.2.5.1 Автомат с магазинной памятью

$$P_M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F),$$

где Q – конечное множество состояний автомата;

Σ – конечный входной алфавит;

Γ – конечное множество магазинных символов;

$\delta\ (q, ck, zj)$ – функция переходов;

$q_0 \in Q$ – начальное состояние автомата;

$z_0 \in \Gamma$ – символ, находящийся в магазине в начальный момент,

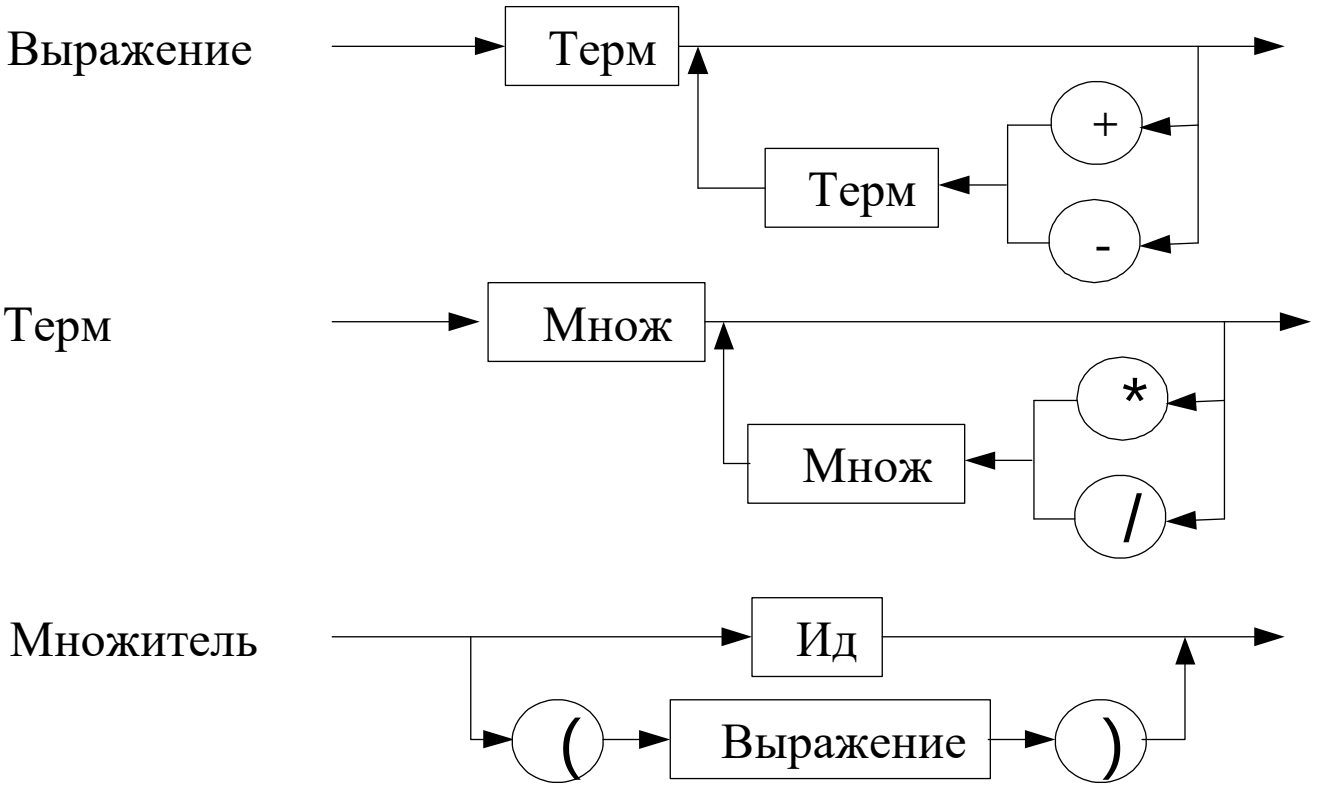
$F \subseteq Q$ – множество заключительных (допускающих) состояний.

Синтаксические диаграммы выражения

Пример. Синтаксический анализатор выражений.

Σ = {<Ид>, +, –, *, /, (,), ◄, ►}.

Например: **A * (B + C) + (D + F) / (A + B) - C * D**



Объединенная синтаксическая диаграмма

Выражение

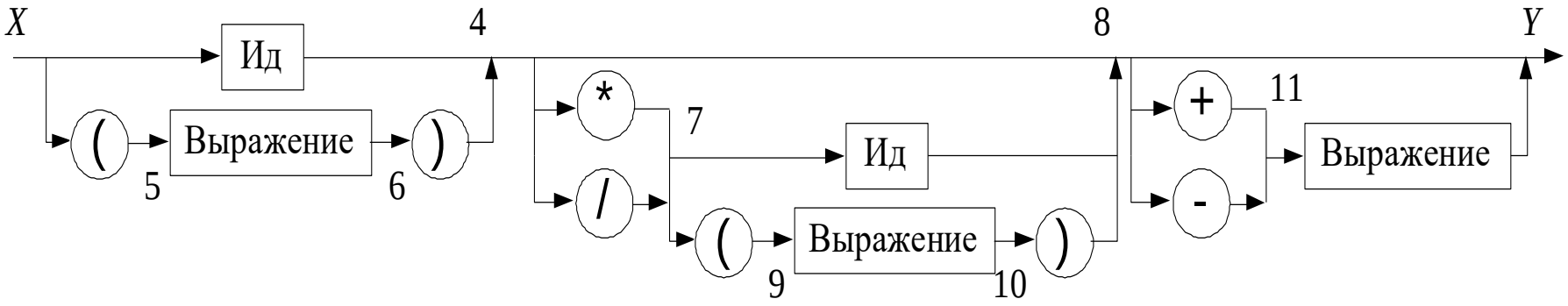


Таблица переходов автомата

	<Ид>	+	-	*	/	(	)	▶	◀
1								2	
2	↓S=3					↓ S =3			
3									К

	<Ид>	+	-	*	/	(	)	▶	◀
X	4					5			
4		11	11	7	7		↑ S		↑ S
5	↓ S =6					↓ S =6			
6							4		
7	8					9			
8		11	11				↑ S		↑ S
9	↓ S =10					↓ S =10			
10							8		
11	↓ S =Y					↓ S =Y			
Y							↑ S		↑ S

Алгоритм распознавателя

```
q:=1
Ind := 1
Mag :=∅
Цикл-пока q ≠ «Е» и q ≠ «К»
    q := Table [q, String[Ind]].q
    Если q = '↓'
     то    Mag↓ Table [q, String[Ind]].S
           q := X
     иначе Если q = '↑'
              то Mag ↑q
              иначе Ind := Ind +1
           Все-если
    Все-если
Все-цикл
Если q = «К»
    то «Строка принята»
    иначе «Строка отвергнута»
Все-если
```

4.2.5.2 Синтаксические анализаторы LL(k) грамматик. Левосторонний нисходящий грамматический разбор

Распознано	Строка	Правила	Правило
	–45◄	<Целое> ◄	
	–45◄	<Знак><ЦБЗ>◄	1а – ?
	–45◄	+<ЦБЗ>◄	4а – нет
		–<ЦБЗ>	4б – да
–	45◄	<ЦБЗ>◄	
–	45◄	<Цифра><ЦБЗ>◄	2а – ?
–	45◄	0..3<ЦБЗ>◄	3а..3г – нет
		4<ЦБЗ>◄	3е – да
–4	5◄	<ЦБЗ>◄	
–4	5◄	<Цифра><ЦБЗ>◄	2а – ?
–4	5◄	0..4<ЦБЗ>◄	3а..3е – нет
		5<ЦБЗ>◄	3ж – да
–45	◄	<ЦБЗ>◄	
–45	◄	<Цифра><ЦБЗ>◄	2а –?
–45	◄	0..9<ЦБЗ>◄	3а..3и – нет
	◄	<ЦБЗ>◄	2б –?
	◄	<Цифра>◄	3а..3и – нет
–4	5◄	<Цифра>◄	2б – ?
	5◄	1..4<ЦБЗ>◄	3а..3е – нет
		5<ЦБЗ>◄	3ж – да
–45	◄	◄	Конец

Возвраты вызваны неправильным выбором альтернатив правил

В LL(k) грамматиках обеспечивается однозначный выбор правил в процессе разбора

Синтаксические анализаторы LL(k) грамматик (2)

Пример. Дана грамматика записи выражений:

1) <Строка> ::= <Выр>◄

2) <Выр> ::= <Терм> <Слож>

3) <Слож> ::= e | + <Терм> <Слож> | - <Терм><Слож>

4) <Терм> ::= <Множ><Умн>

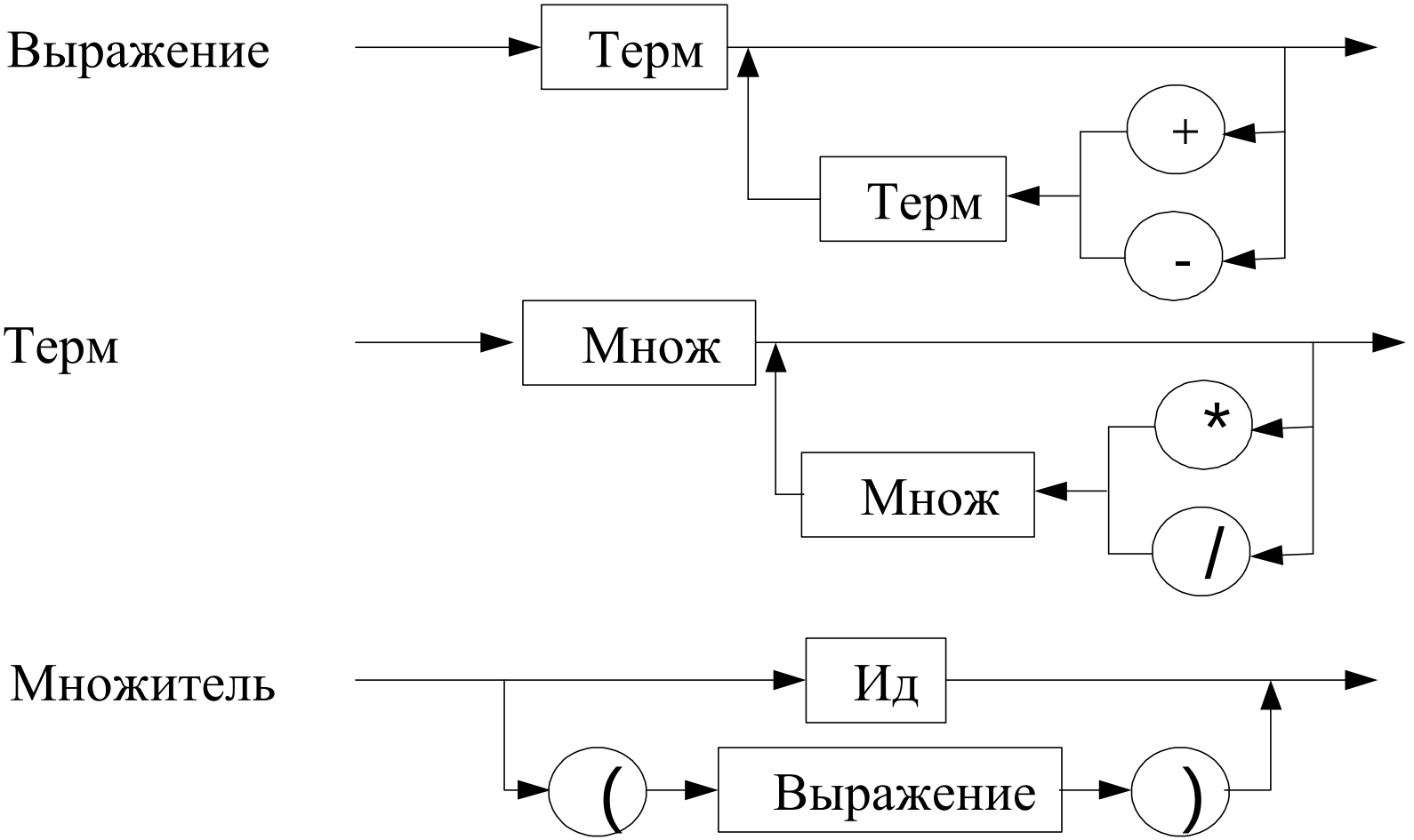
5) <Умн> ::= e | *<Множ><Умн> | / <Множ><Умн>

6) <Множ> ::= <Ид> | (<Выр>)

Распознаваемая строка	Стек	Правило
<Ид>*(<Ид>+<Ид>)+<Ид>◄	<Выр> ◄	2
<Ид>*(<Ид>+<Ид>)+<Ид>◄	**<Терм>**<Слож>◄	4
<Ид>*(<Ид>+<Ид>)+<Ид>◄	<**Множ**><Умн><Слож>◄	6а
<Ид>*(<Ид>+<Ид>)+<Ид>◄	<Ид><Умн><Слож>◄	Удалить символ
***(<Ид>+<Ид>)+<Ид>◄	<**Умн**><Слож>◄	5б
***(<Ид>+<Ид>)+<Ид>◄	*<Множ><Умн><Слож>◄	Удалить символ
(<Ид>+<Ид>)+<Ид>◄	<**Множ**><Умн><Слож>◄	6б
(<Ид>+<Ид>)+<Ид>◄	(<Выр>)<Умн><Слож>◄	Удалить символ

Метод рекурсивного спуска

Пример. Выражение



Алгоритм распознавателя

Функция Выражение:Boolean:

R:=Терм()

Цикл-пока R=true и (NextSymbol =’+’ или NextSymbol =’-’)

R:=Терм()

Все-цикл

Выражение:= R

Все

Функция Терм:Boolean:

Множ()

Цикл-пока R=true и (NextSymbol =’*’ или NextSymbol =’/’)

R:=Множ()

Все-цикл

Терм:= R

Все

Алгоритм распознавателя (2)

Функция Множ:Boolean:
Если NextSymbol =’(’
то R:=Выражение()
Если NextSymbol ≠ ‘)’
то Ошибка Все-если
иначе R:= Ид()
Все-если
Все
Основная программа:
R:=Выражение()
Если NextSymbol ≠ ‘◄’
то Ошибка ()
Все-если
Конец

Текст программы

```
program Compiler;
{$APPTYPE CONSOLE}
uses  SysUtils;
Type SetofChar=set of AnsiChar;
Const Bukv:setofChar=['A'..'Z','a'..'z'];
Const Cyfr:setofChar=['0'..'9'];
Const Razd:setofChar=[' ','+','-','*','/',')'];
Var St:shortstring; R:boolean;

Function Culc(Var St:shortstring;Razd:setofChar):boolean;forward;

Procedure Error(St:shortstring); {вывод сообщений об ошибках}
  Begin      WriteLn('Error *** ', st, ' ***');  End;

Procedure Probel(Var St:shortstring); {удаление пробелов}
  Begin   While (St<>'') and (St[1]=' ') do Delete(St,1,1);
  End;
```

Текст программы (2)

```
Function Id(Var St:shortstring; Razd:setofChar):boolean;
 Var S:shortString; State,Ind,Col:byte;
 Const TableId:array[1..2,1..4] of Byte=
                    ((2,0,0,0),(2,2,10,0));
 Begin
   Probel(St); {процедура удаления пробелов}
   State:=1;   S:='';
   while (State<>0) and (State<>10) and (Length(St)<>0)
     do
      begin
        if St[1] in ['A'..'Z','a'..'z'] then Col:=1
        else  if St[1] in ['0'..'9'] then Col:=2
              else if (St[1] in Razd) then Col:=3
                   else Col:=4;
        State:=TableId[State,Col];
```

Текст программы (3)

```
 if (State<>0) and (State<>10) then {правильный символ}
   begin  S:=S+St[1];  Delete(St,1,1);  end;
end;
if length(st)=0 then State:=10;  {строка закончилась}
if (State=10) and (S<>'') then   {правильно}
  Begin  Result:=true;  WriteLn('Identify=',S); End
else
  if (State=0) then              {ошибка символа}
    Begin  Result:=false;
           WriteLn('Wrong symbol *',St[1],'*');
    End
  else                           {нет идентификатора}
    Begin  Result:=false;
           WriteLn('Identifier waits...', St);
    End;
End;
```

Текст программы (4)

```
Function Mult(Var St:shortstring;Razd:setofChar):boolean;
  Var R:boolean;
  Begin
    Probel(St);
    if St[1]='(' then
      begin
        Delete(St,1,1); Probel(St);
        R:=Culc(St,Razd);
        Probel(St);
        if R and (St[1]=')') then
                Delete(St,1,1) else Error(St);
      end
    else R:=Id(St,Razd);
    Mult:=R;
  End;
```

Текст программы (5)

```
Function Term(Var St:shortstring;Razd:setofChar):boolean;
  Var R:boolean;
  Begin
     R:=Mult(St,Razd);
     if R then
       begin
         Probel(St);
         While ((St[1]='*') or (St[1]='/')) and R do
           begin
             Delete(St,1,1);
             R:=Mult(St,Razd);
           end;
       end;
     Term:=R;
  End;
```

Текст программы (6)

```
Function Culc(Var St:shortstring;Razd:setofChar):boolean;
  Var R:boolean;
  Begin
     R:=Term(St,Razd);
     if R then
       begin
         Probel(St);
         While ((St[1]='+') or (St[1]='-')) and R do
           begin
             Delete(St,1,1);
             R:=Term(St,Razd);
           end;
       end;
     Culc:=R;
  End;
```

Текст программы (6)

```
Begin
  Writeln('Input Strings:'); Readln(St);
  R:=true;
  While (St<>'end') and R do
    Begin
      R:=Culc(St,Razd);
      if R and (length(st)=0) then Writeln('Yes')
                              else Writeln('No');
      Writeln('Input Strings:'); Readln(St);
    End;
    writeln('Press Enter');
    readln;
End.
```

Результат работы программы

```
Input Strings:
tyy+(hjh-hj)*hj
Identify=tyy
Identify=hjh
Identify=hj
Identify=hj
Yes
Input Strings:
end
```

4.2.5.3 Синтаксические анализаторы LR(k) грамматик. Грамматики предшествования

Левосторонний восходящий грамматический разбор :

-45

<знак> 45

<знак> <цифра>5

<знак> <цбз>5

<целое> 5 — **Тупик!**

<знак> <цбз>5

<знак><целое> 5 — **Тупик!**

<знак> <цбз>5

<знак> <цбз><цифра>

<целое> <цифра> — **Тупик!**

<знак> <цбз><цифра>

<знак> <целое> <цифра> — **Тупик!**

<знак> <цбз><цифра>

<знак> <цбз>

<целое>

<целое> ::= <знак><ЦБЗ>|< ЦБЗ >,
<ЦБЗ> ::= <ЦБЗ><цифра>|<цифра>,
<цифра > ::= 0|1|2|3|4|5|6|7|8|9,
<знак> ::= +| -

Возвраты возникают из-за неверного выбора основы

Грамматики LR(k) обеспечивают однозначный выбор основы

Грамматики предшествования

....αβ....

Если два символа α, β $\in V$ расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные отношениями предшествования:

1) α принадлежит основе, а β – нет, т. е. α – конец основы: α ·> β;
2) β принадлежит основе, а α – нет, т. е. β – начало основы: α <· β ;
3) α и β принадлежит одной основе, т. е. α =· β;
4) α и β не могут находиться рядом в сентенциальной форме (ошибка).

Грамматикой предшествования называется грамматика, в которой из последовательности символов однозначно следует определение основы.

Грамматикой операторного предшествования называется грамматика, в которой существует однозначное отношение предшествования терминальных символов, которое не зависит от нетерминальных символов, находящихся между ними.

Построение таблицы предшествования

Пример. Грамматика арифметических выражений (с левосторонней рекурсией):

<Выражение> ::= <Терм>|<Выражение>+<Терм>|<Выражение>- <Терм>
<Терм> ::= <Множитель> | <Терм>* <Множитель> | <Терм> / <Множитель>
<Множитель> ::= (<Выражение>) | <Идентификатор>

<· - начало основы;
·> - конец основы;
= - одна основа;
? - ошибка

►A+ ► A* ►(A ► A) ►A◄
+A+ +A* +(A +A) +A◄
*A+ *A* *(A *A) *A◄
(A+ (A* ((A (A) (A◄
A)+ A)* A)(A)) A)◄

	+	*	(	)	◄
►	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

Пример разбора выражения стековым методом

►d+c*(a+b) ◄

Содержимое стека	Анализируемые символы	Отношение	Операция	Тройка	Результат свертки
►	d+	<·	Перенос		
►d+	c*	<·	Перенос		
►d+ c*	(	<·	Перенос		
►d+ c*(	a+	<·	Перенос		
►d+ c*(a+	b)	·>	Свертка	R_1:= a + b	<Выражение>
►d+ c*(	R_1)	=·	Свертка	R_1:= (R_1)	<Множитель>
►d+ c*	R_1◄	·>	Свертка	R_2:= c* R_1	<Терм>
►d+	R_2◄	·>	Свертка	R_3:= d+ R_2	<Выражение>
►	R_3◄	Конец			

Пример.

Разработать программу, осуществляющую лексический анализ идентификаторов и служебных слов, а также синтаксический анализ сравнений (не более одной операции сравнения вида =, <>, >, <, >=, <=), выражений с операциями +, -, *, / и скобками, операторов условной передачи управления и присваивания в синтаксисе языка Паскаль. Например:

```
if aaaa>vvvv then j:=hhhh
else if h then ffff:=hhh+(ppp+yyy);
```

Описание языка:

<Оператор> ::= <Оператор if> ; | <Присваивание> ;

<Оператор if> ::=if <Условие> then <Оператор> else <Оператор>|
if <Условие> then <Оператор>

<Присваивание> ::= <Идентификатор> := <Выражение>

<Условие> ::= <Идентификатор> = <Идентификатор>|
<Идентификатор> > <Идентификатор>|
<Идентификатор> < <Идентификатор> …

Лексический анализ

Токены: V_ – идентификатор - операнд;

@@ – пустой операнд – дополнительный токен, который позволяет при разборе считывать строго по два токена <операнд-оператор>;

if – служебное слово if;

th – служебное слово then;

el – служебное слово else;

>_, <_, =_, <>, >=, <= – операции сравнения;

+_, -_, *_, /_, (_,)_ – операторы выражения;

:= - служебное слово «присвоить»;

;_ - конец оператора.

Для примера, приведенного в задании:

```
if aaaa>vvvv then j:=hhhh
else if h then ffff:=hhh+(ppp+yyy);
```

результат работы сканера должен выглядеть так:

```
@@ if V_ >_ V_ th V_ := V_
   el @@ if V_ th V_ := V_ *_ @@ (_ V_ +_ V_ )_ ;_
```

Таблица предшествования

	=	+	*	(	)	:=	If	Th	El	;	??
#	E	E	E	E	E	<.	<.	E	E	K	E
=	E	E	E	E	E	E	E	>.	E	E	E
+	E	>.	<.	<.	>.	E	E	E	E	>.	E
*	E	>.	>.	<.	>.	E	E	E	E	>	E
(	E	<.	<.	<.	()	E	E	E	E	E	E
:=	E	<.	<.	<.	E	E	E	E	>.	>.	E
If	<.	E	E	E	E	E	E	=.	E	E	E
Th	E	E	E	E	E	<.	<.	E	=.	>.	E
El	E	E	E	E	E	<.	<.	E	>.	>.	E

При реализации использовано следующее кодирование:

<.– начало основы;

=. – середина основы;

>. – конец основы;

() – скобки;

5 – выход;

50 – ошибка.

Лексический анализ

Input Strings:

```
if aaaa>vvvv then j:=hhhh else if h then
                                        ffff:=hhh*(ppp+yyy);
Slugeb slovo if
Identify=aaaa
Slugeb simbol >
Identify=vvvv
Slugeb slovo then
Identify=j
……
```

After Scan:

```
@@ifV > V thV :=V el@@ifV thV :=V * @@( V + V ) ;
```

Результат синтаксического анализа

```
if aaaa>vvvv then j:=hhhh
             else if h then ffff:=hhh+(ppp+yyy);
```

After Scan:

```
@@ifV > V thV :=V el@@ifV thV :=V * @@( V + V ) ;
```

Comands: `V > V` // сравнение

Comands: `V :=V` // присваивание в ветви «да» внешнего if

Comands: `V + V` // сложение

Comands: `V * C` // умножение

Comands: `V :=C` // присваивание в ветви «да» вложенного if

Comands: `ifV thOp` // вложенный if (без else)

Comands: `ifL thOpelOI` // внешний if

Yes

4.2.6 *Польская запись. Алгоритм Бауэра-Замельзона*

Польская запись представляет собой последовательность команд двух типов:

K_I, где I – идентификатор операнда – выбрать число по имени I и заслать его в стек операндов;

K_ξ, где ξ – операция – выбрать два верхних числа из стека операндов, произвести над ними операцию ξ и занести результат в стек операндов.

Пример:

$A+B*C \Rightarrow K_A K_B K_C K_* K_+$

Алгоритм Бауэра-Замельзона

Построение польской записи :

а) если символ – операнд, то вырабатывается команда $К_I$,

б) если символ – операция, то выполняются действия согласно таблице:

η \ ξ	+	*	(	)	◄
►	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

η\ξ	+	*	(	)	←
→	I	I	I	?	Вых
+	II	I	I	IV	IV
*	IV	II	I	IV	IV
(	I	I	I	III	?

Операции:

I – заслать ξ в стек операций и читать следующий символ;

II – генерировать **Kη** , заслать ξ в стек операций и читать следующий символ;

III – удалить верхний символ из стека операций и читать следующий символ;

IV – генерировать **Kη** и повторить с тем же входным символом.

Пример. Построить тройки для ***(a+b*c)/d*.**

Построение польской записи:

Стек операций	Символ	Действие	Команда
▶	(	I	
▶ (	a		$\mathbf{K_a}$
▶ (	+	I	
▶ (+	b		$\mathbf{K_b}$
▶ (+	*	I	
▶ (+ *	c		$\mathbf{K_c}$
▶ (+ *	)	IV	$\mathbf{K_*}$
▶ (+	)	IV	$\mathbf{K_+}$
▶ (	)	III	
▶	/	I	
▶ /	d		$\mathbf{K_d}$
▶ /	←	IV	$\mathbf{K_/}$
▶	←	Конец	

$\mathbf{K_a\ K_b\ K_c\ K_*\ K_+\ K_d\ K_/}$ или **abc*+d/**

Пример (2)

Выполнение операций польской записи:

Стек операндов	Команда	Тройка
∅	K_a	
a	K_b	
a b	K_c	
a b c	K_*	$T_1 = b*c$
a T_1	K_+	$T_2 = a+T_1$
T_2	K_d	
T_2 d	$K_/$	$T_3 = T_2/d$
T_3		

4.3 Распределение памяти под переменные

- **статическое** – выполняется в процессе компиляции или загрузки в память (для сегмента неинициированных данных), используется для хранения глобальных переменных;
- **автоматическое** – выполняется при вызове подпрограмм, используется для локальных переменных, размещаемых в стеке;
- **управляемое** – выполняется по запросу программиста (new, delete), используется для динамических переменных, размещаемых в динамической памяти;
- **базированное** – также выполняется по запросу программиста, но большими фрагментами (getmem, freemem), за размещение переменных отвечает программист.. .

4.4 Генерация и оптимизация кодов

Пример «заготовки»:

```
Mov    AX, <Op1>
Add    AX, <Op2>
Mov   <Result>, AX
```

Машинно независимая оптимизация включает:

а) исключение повторных вычислений одних и тех же операндов;

б) выполнение операций над константами во время трансляции;

в) вынесение из циклов вычисления величин, не зависящих от параметров циклов;

г) упрощение сложных логических выражений и т. п.

Машинно зависимая оптимизация включает:

а) исключение лишних передач данных типа «память-регистр»;

б) выбор более эффективных команд т. п.