2021

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. Проблема трансляции выражений

Пусть задано выражение: $A = x*(y^2-1)+(x*y-k)*h$

Для автоматического составления кода:

1) необходимо построить "тройки";
2) построить код выполнения тройки.

Тройка – это элементарное выражение, включающее два операнда и операцию.

Из заданного выражения можно построить следующие тройки:

y ^ 2 => T1

T1 – 1 => T2

x * T2 => T3

x * y => T4

T4 – k => T5

T5 * h => T6

T3+T6 = T7

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.

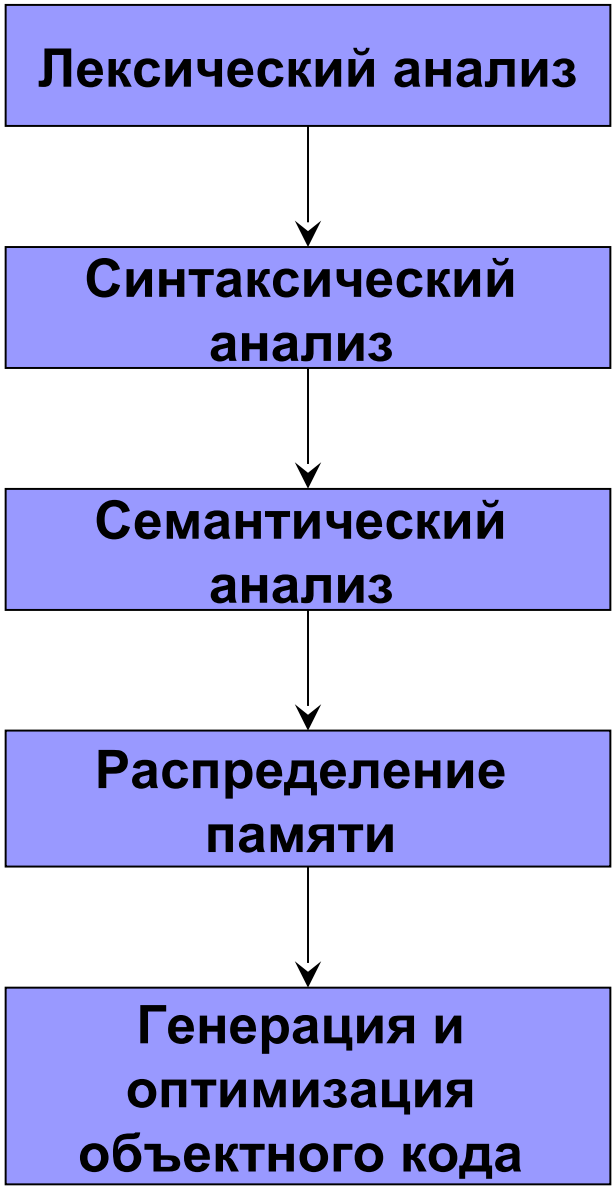
Макрогенератор (для компиляторов – ***препроцессор***) – программа, которая обрабатывает исходную программу, как текст, и выполняет в нем замены указанных символов на подстроки. Макрогенератор обрабатывает программу до трансляции.

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



Процесс выделения отдельных слов (*лексем*) в предложениях языка и преобразования этих предложений в строку однородных символов - *токенов*.

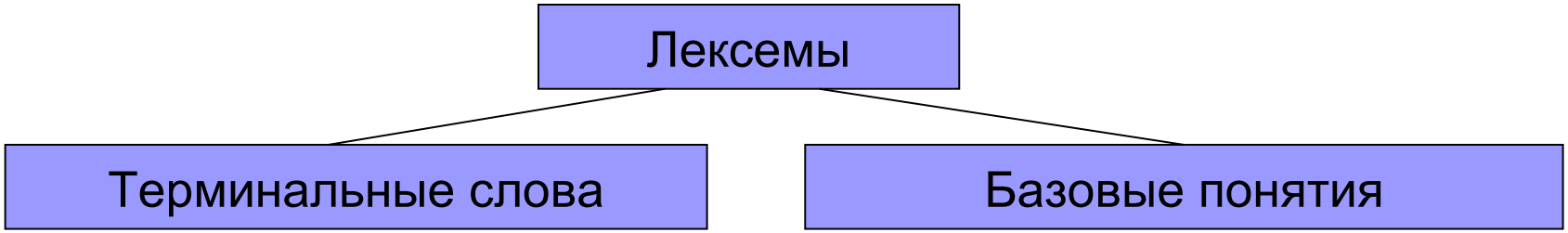
Процесс распознавания конструкций языка в строке токенов.

Процесс распознавания/проверки смысла конструкций.

Процесс назначения адресов для размещения именованных и неименованных констант, а также переменных программы.

Процесс формирования семантически эквивалентной исходной программе программы на выходном языке.

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 e, 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

Рекурсия позволяет генерировать числа любой длины больше 1!

Форма Бэкуса-Наура (ФБН)

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> ::= +| -

Расширенная форма Бэкуса-Наура (РФБН)

Условные обозначения:

Имя – нетерминальный символ – конструкция;

"Имя" или 'Имя' – терминальный символ – символ алфавита;

= – «это есть» - разделитель левой и правой частей правила;

, – конкатенация (сцепление) символов в строке;

[…] – условное вхождение, указание необязательной части;

{…} – повтор;

| - выбор (или);

(…) – группировка символов.

Пример:

Целое = ["+"|"-"] Цифра{Цифра}.
Цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

4.2.2 *Грамматический разбор*

Грамматический разбор - процесс сопоставления линейной последовательности лексем (слов, токенов) языка его формальной грамматике. Позволяет определить принадлежность предложения языку.

Вывод – последовательность подстановок.

Дерево грамматического разбора (синтаксическое дерево) – графическое представление вывода

Пример. Вывод строки «-45»:

<целое> $\Rightarrow_1$

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

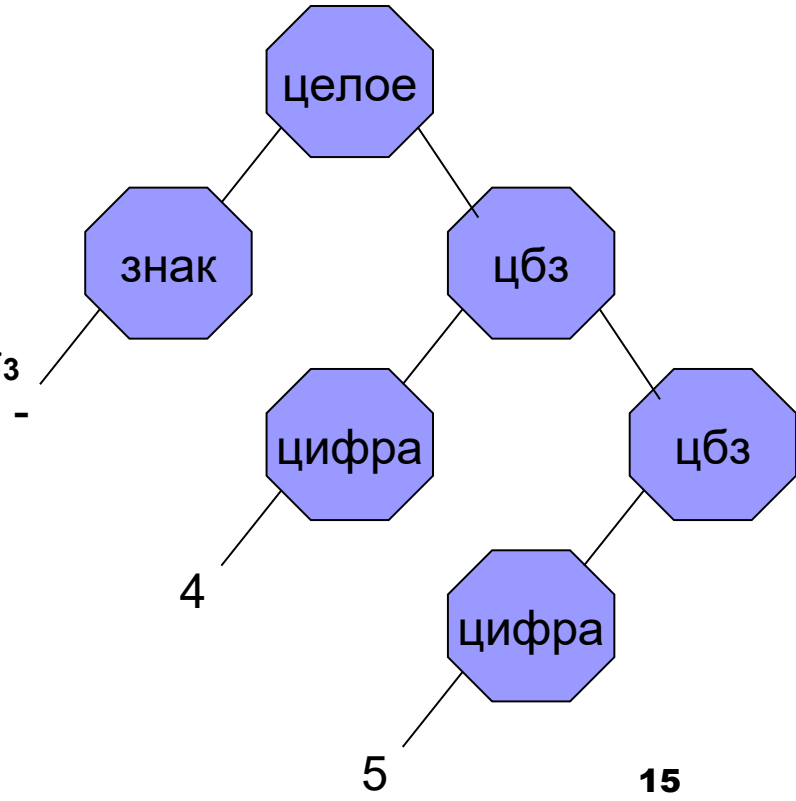
$\Rightarrow_2$ **<знак><цифра><целое без знака>** $\Rightarrow_3$

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

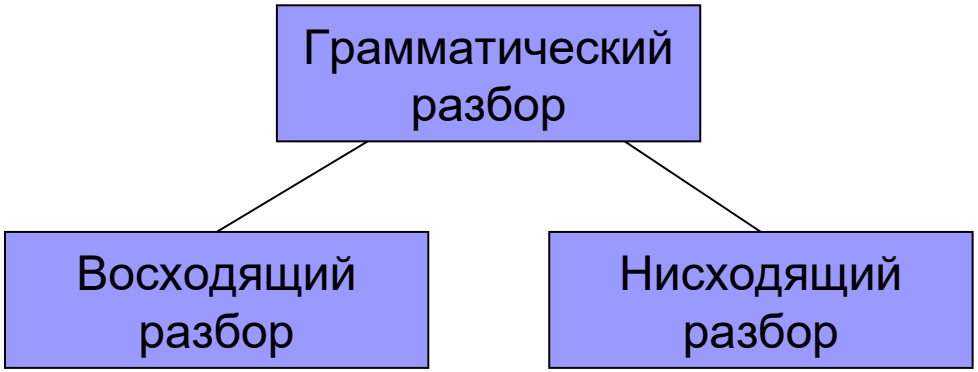
$\Rightarrow_4$ **- <цифра><цифра>** $\Rightarrow_5$

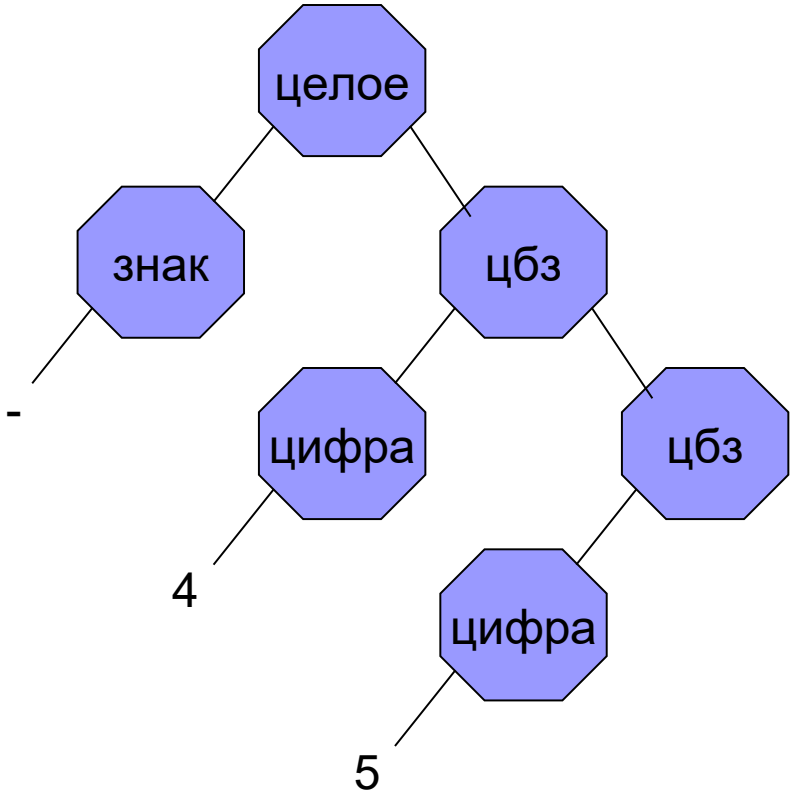
$\Rightarrow_5$ **- 4<цифра>** $\Rightarrow_6$

$\Rightarrow_6$ **- 45**



Виды грамматического разбора





Восходящий разбор – от символов или лексем языка к конструкциям (понятиям более высокого порядка, аксиоме).

Нисходящий разбор – от конструкции (понятия более высокого порядка, аксиомы) к символам или лексемам языка.

1. Левосторонний нисходящий грамматический разбор (разбор "сверху вниз")

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

а) **<Целое> ::= <Знак><Целое без знака>|<Целое без знака>**

б) **<Целое без знака> ::= <Цифра><Целое без знака>|<Цифра>,**

в) **<Цифра > ::= 0|1|2|3|4|5|6|7|8|9,**

г) **<Знак> ::= +| - .**

Правостороння я рекурсия

Альтернативные правила нумеруем цифрами 1, 2, 3 ...

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 – нетерминал, то заменяем его на первое из правил, его определяющих.

И так далее…

Таблица грамматического разбора левосторонним нисходящим методом

№ шага	Распо-знано	Распозна-ваемая строка	Строка правил	Действие
1		-45	<Целое>	Подстановка правила а1
2		-45	<Знак><ЦБЗ>	Подстановка правила г1
		-45	+<ЦБЗ>	Ошибка, возврат к шагу 2
3		-45	<Знак><ЦБЗ>	Подстановка правила г2
		-45	- <ЦБЗ>	**Символ распознан**
4	-	45	<ЦБЗ>	Подстановка правила б1
5	-	45	<Цифра><ЦБЗ>	Подстановка правила в1
	-	45	0 <ЦБЗ>	Ошибка, возврат к шагу 5
6	-	45	<Цифра><ЦБЗ>	Подстановка правила в2
	-	45	1 <ЦБЗ>	Ошибка, возврат к шагу 6
7	-	45	<Цифра><ЦБЗ>	Подстановка правила в3
	-	45	2 <ЦБЗ>	Ошибка, возврат к шагу 7
8	-	45	<Цифра><ЦБЗ>	Подстановка правила в4
	-	45	3 <ЦБЗ>	Ошибка, возврат к шагу 8
9	-	45	<Цифра><ЦБЗ>	Подстановка правила в5
	-	45	4 <ЦБЗ>	**Символ распознан**

а) **<Целое> ::= <Знак><ЦБЗ>|< ЦБЗ >**

б) **< ЦБЗ > ::= <Цифра>< ЦБЗ >,**

в) **<Цифра > ::= 0|1|2|3|4|5|6|7|8|9,**

г) **<Знак> ::= +| - .**

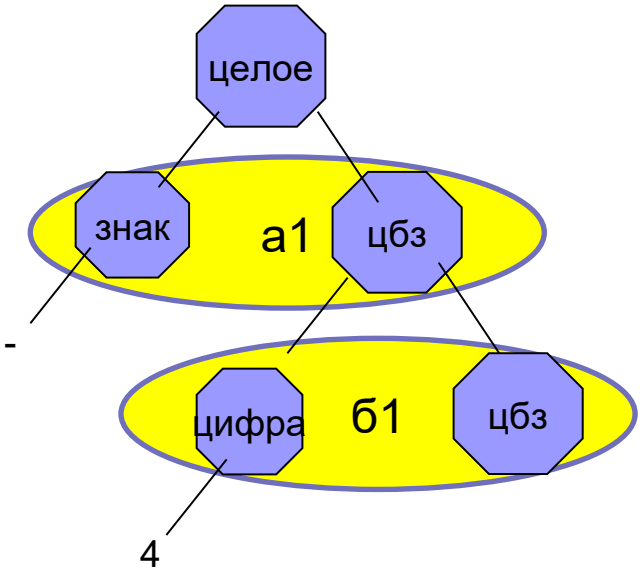


Таблица грамматического разбора (2)

№ шага	Распо-знано	Распозна-ваемая строка	Строка правил	Действие
10	-4	5	<**ЦБЗ**>	Подстановка правила б1
11	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в1
	-4	5	0 <ЦБЗ>	Ошибка, возврат к шагу 11
12	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в2
	-4	5	1 <ЦБЗ>	Ошибка, возврат к шагу 12
13	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в3
	-4	5	2 <ЦБЗ>	Ошибка, возврат к шагу 13
14	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в4
	-4	5	3 <Цбз>	Ошибка, возврат к шагу 14
15	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в5
	-4	5	4 <Цбз>	Ошибка, возврат к шагу 15
16	-4	5	<**Цифра**><ЦБЗ>	Подстановка правила в6
	-4	5	5 <ЦБЗ>	**Символ распознан**
17	-45	∅	<**ЦБЗ**>	Подстановка правила б1
18.. 27	-45	∅	<**Цифра**><ЦБЗ>	Подстановки правил в1-в10
	-45	∅	0 .. 9 <ЦБЗ>	Ошибки, возвраты, возврат к шагу 17

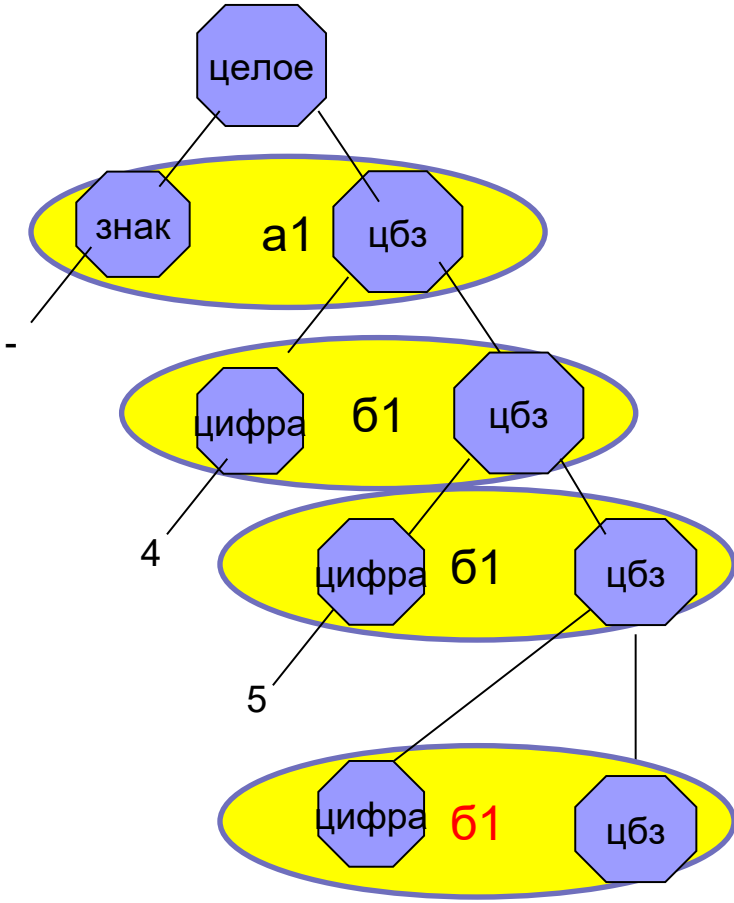
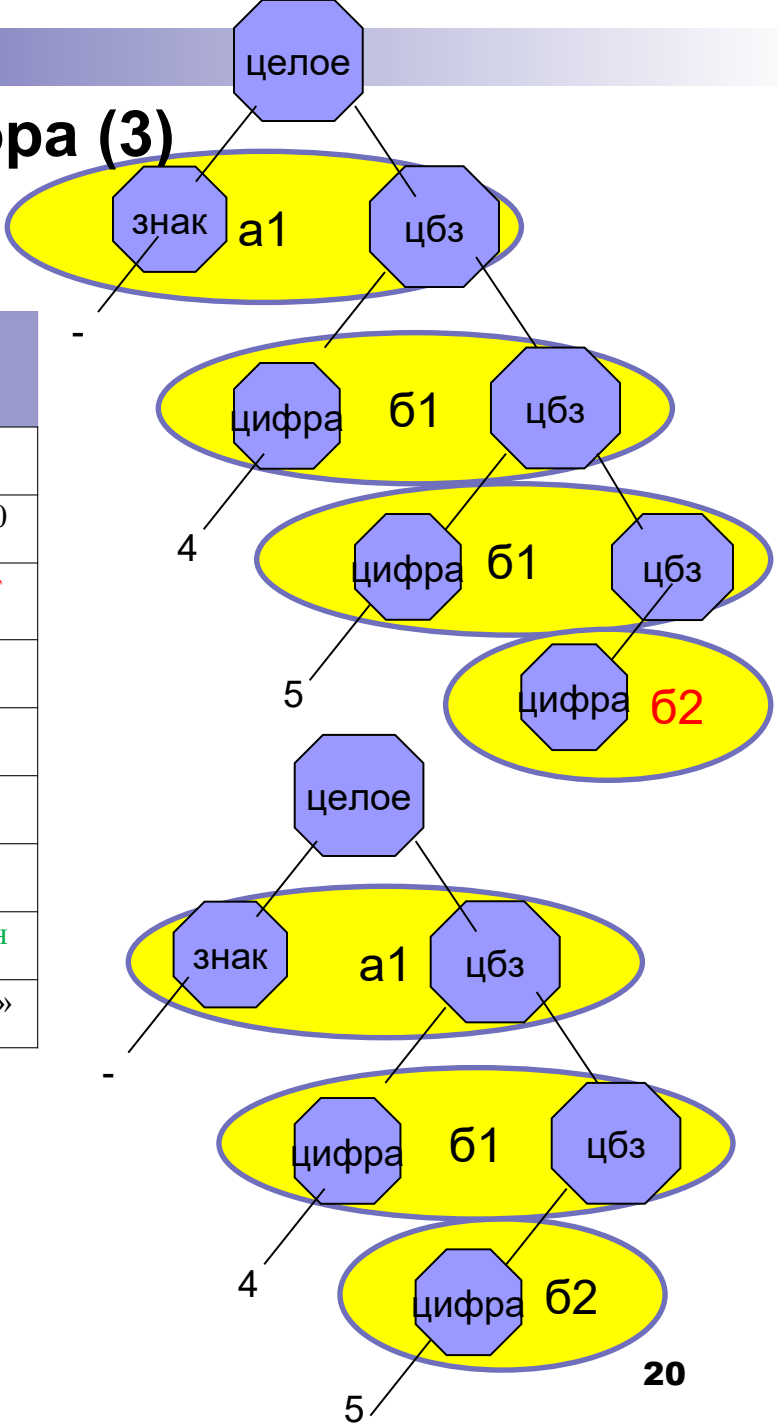


Таблица грамматического разбора (3)

№ шага	Распо-знано	Распозна-ваемая строка	Строка правил	Действие
28	-45	∅	<**ЦБЗ**>	Подстановка правила б2
29..38	-45	∅	<**Цифра**>	Подстановки правил в1-в10
	-45	∅	0 .. 9	Ошибки, возвраты, возврат к шагу 10
39	-4	5	<**ЦБЗ**>	Подстановка правила б2
40..44	-4	5	<**Цифра**>	Подстановки правил в1-в5
	-4	5	0 .. 4	Ошибки и возвраты
45	-4	5	<**Цифра**>	Подстановка правила в6
	-4	5	5	Символ распознан и удален
46	-45	∅	∅	Конец «Строка распознана»



Возвраты возникают из-за неверного выбора правила!!!

2. Левосторонний восходящий грамматический разбор

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

а) <Целое> ::= <Знак><Целое без знака>|<Целое без знака>,

б) <Целое без знака> ::= <Целое без знака><Цифра>|<Цифра>,

в) <Цифра > ::= 0|1|2|3|4|5|6|7|8|9,

г) <Знак> ::= +| - .

Левосторонняя рекурсия

Идея метода:

При разборе строка 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 и поиска другой основы.

Таблица грамматического разбора

№ шага	Распознаваемая строка	Основа	Операция
1	-45	-	Свертка по правилу г1
2	**<Знак>** 45	<Знак>	Нет правила для свертки. Формирование следующей основы
3	**<Знак> 4**5	<Знак> 4	Нет правила для свертки. Формирование следующей основы
4	<Знак> **45**	**4**	Свертка по правилу в5
5	**<Знак>** <Цифра> 5	<Знак>	Нет правила для свертки. Формирование следующей основы
6	**<Знак> <Цифра>** 5	<Знак> <Цифра>	Нет правила для свертки. Формирование следующей основы
7	<Знак> <**Цифра**> 5	<Цифра>	Свертка по правилу б2
8	**<Знак>** <Цбз> 5	<Знак>	Нет правила для свертки. Формирование следующей основы
9	**<Знак> <Цбз>** 5	<Знак> <Цбз>	Свертка по правилу а1
10	<Целое> 5	Аксиома! Тупик!	**Возврат к шагу 9**. Формирование следующей основы
11	<Знак> **<Цбз>** 5	<Цбз>	Свертка по правилу а2

Таблица грамматического разбора(продолж.)

№	Распознаваемая строка	Основа	Операция
12	<Знак><Целое>5	Аксиома! Тупик!	**Возврат к шагу 11**. Формирование следующей основы
13	**<Знак> <Цбз> 5**	<Знак> **<Цбз>** 5	Нет правила для свертки. Формирование следующей основы
14	<Знак> **<Цбз> 5**	**<Цбз>** 5	Нет правила для свертки. Формирование следующей основы
15	<Знак> <Цбз> **5**	5	Свертка по правилу г6
16	**<Знак> <Цбз>**<Цифра>	<Знак> <Цбз>	Свертка по правилу а1
17	<Целое> <Цифра>	Аксиома! Тупик!	**Возврат к шагу 16**. Формирование следующей основы
18	<Знак> **<Цбз>**<Цифра>	<Цбз>	Свертка по правилу а2
19	<Знак> <Целое> <Цифра>	Аксиома! Тупик!	**Возврат к шагу 18**. Формирование следующей основы
20	<Знак> **<Цбз> <Цифра>**	<Цбз><Цифра>	Свертка по правилу б1
21	**<Знак> <Цбз>**	<Знак> <Цбз>	Свертка по правилу а1
22	<Целое>	Конец	-

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:

$\alpha X \beta \to \alpha x \beta$, где $X \in V_N$, $x \in V_T$, $\alpha, \beta \in V^*$, $V = V_T \cup V_N$, причем α, β одновременно не являются пустыми, а значит возможность подстановки *x* вместо символа *X* определяется присутствием хотя бы одной из подстрок α и β, т. е. *контекста*;

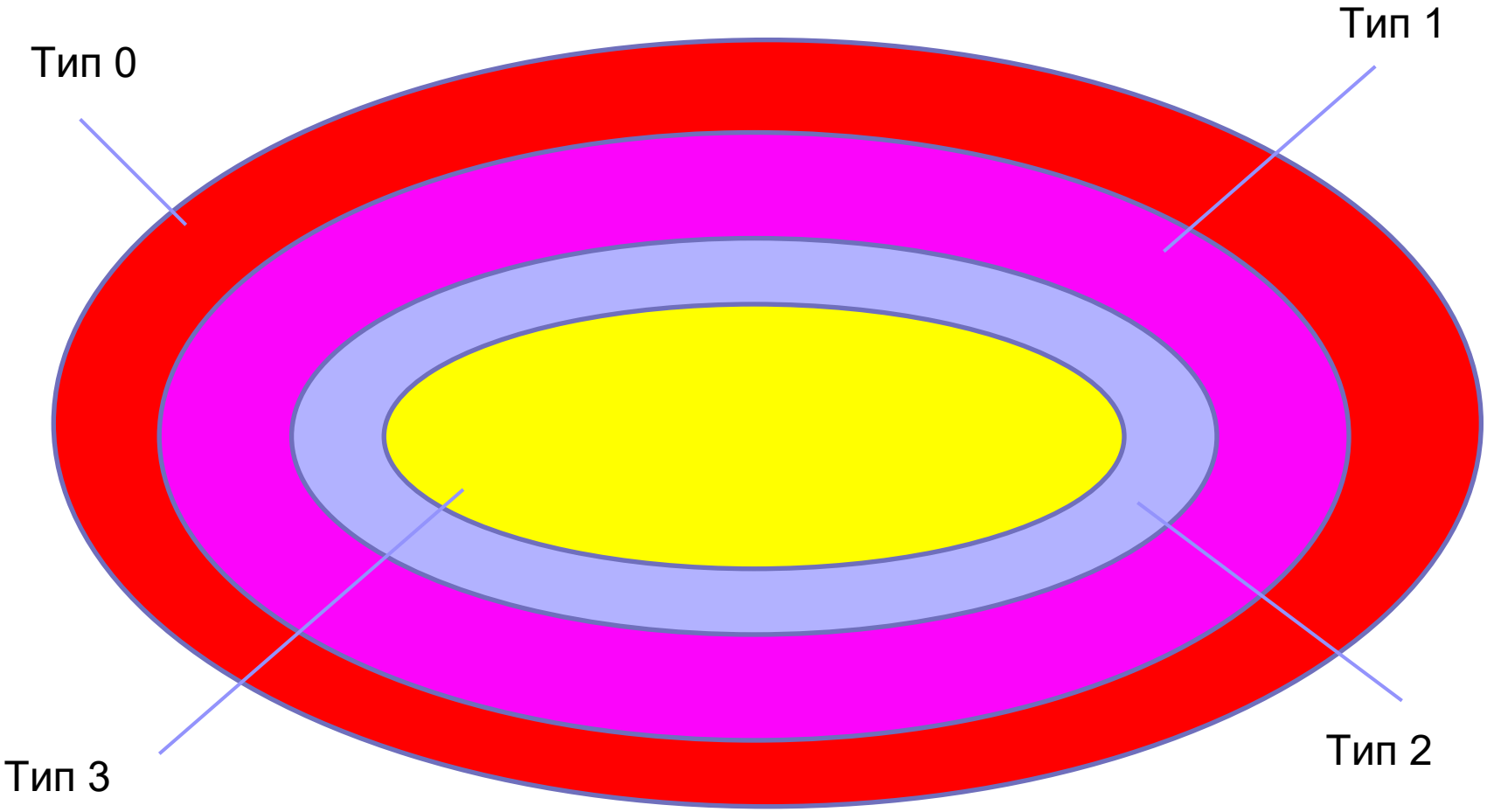
Тип 2 – контекстно-свободные грамматики:

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

Тип 3 – регулярные грамматики:

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

Отношение между грамматиками различных типов



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;

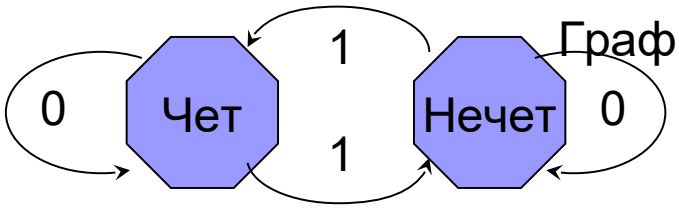
q_0 = Чет;

F = {Чет}

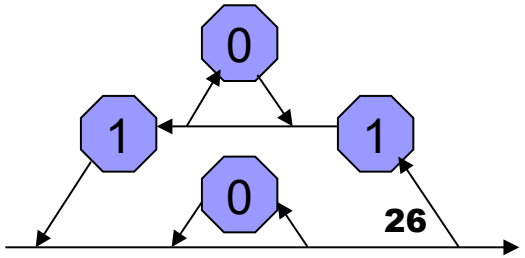
Таблица переходов

	0	1
Чет	Чет	Нечет
Нечет	Нечет	Чет

Граф переходов



Синтаксическая диаграмма



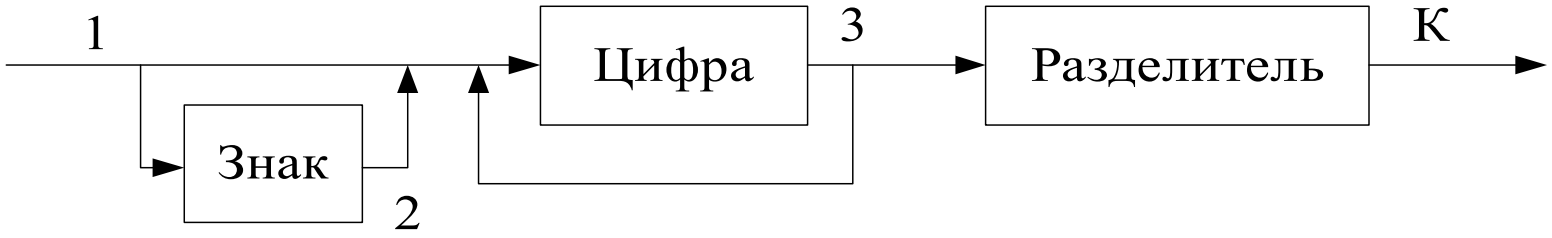
Программная реализация конечного автомата

	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
```

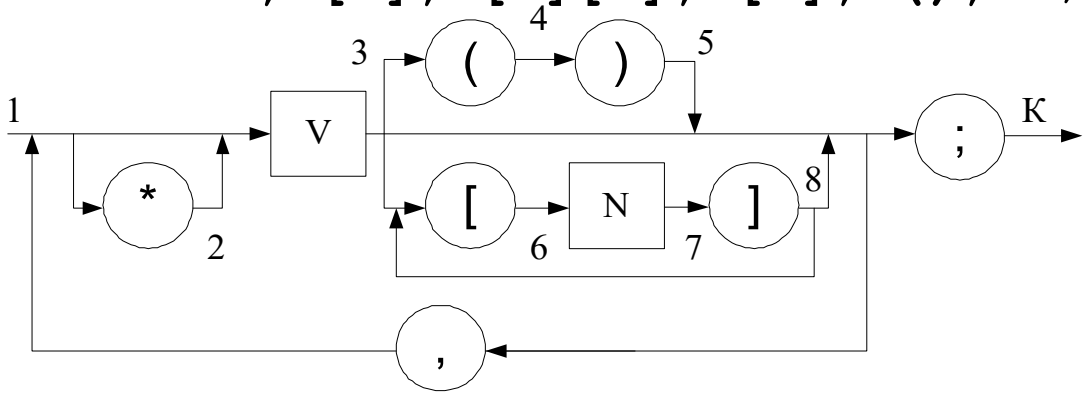
```
Если q = «K»
    то  Выполнить A3
        Вывести «Это число»
    иначе Вывести сообщение Di
Все-если
```

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;
```

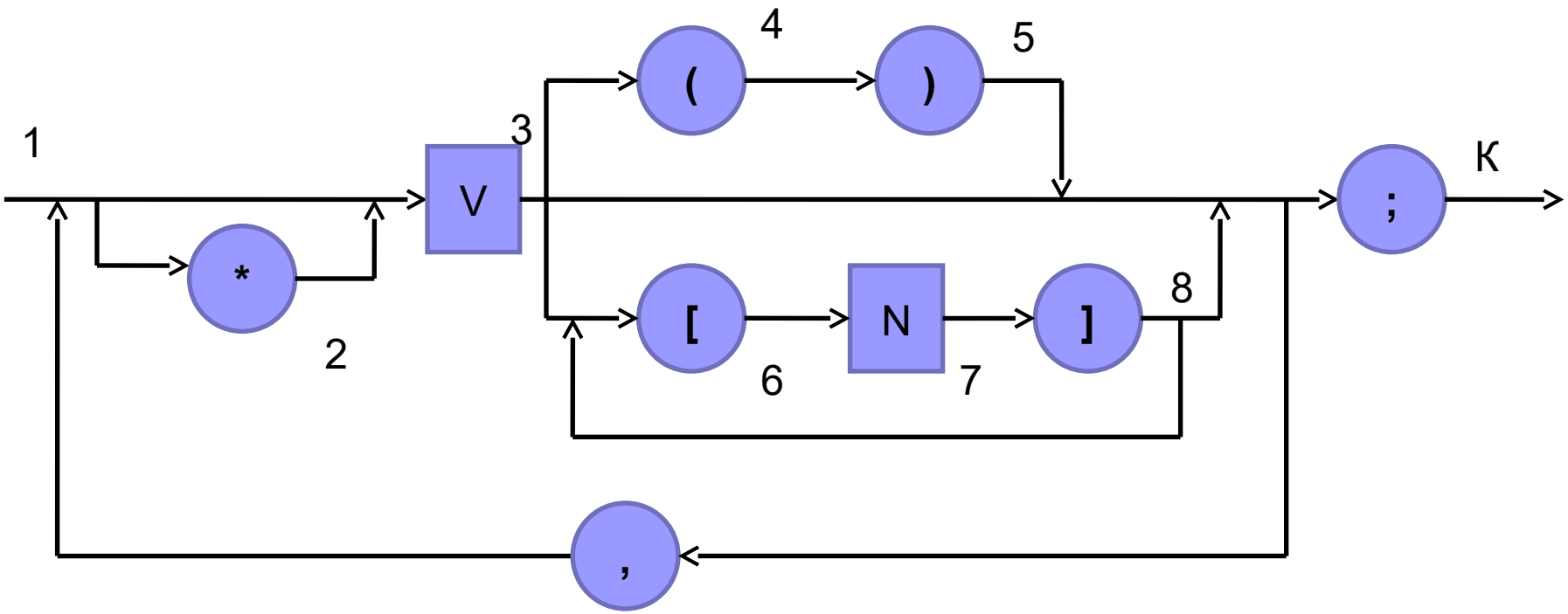


Обозначения:
V – идентификатор;
N – целочисленная константа;
служебные символы: «[ ] ( ) , ; *».

	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

Построение синтаксической диаграммы

```
int V,V[N],V[N][N],V[N],V(),*V;
```



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,

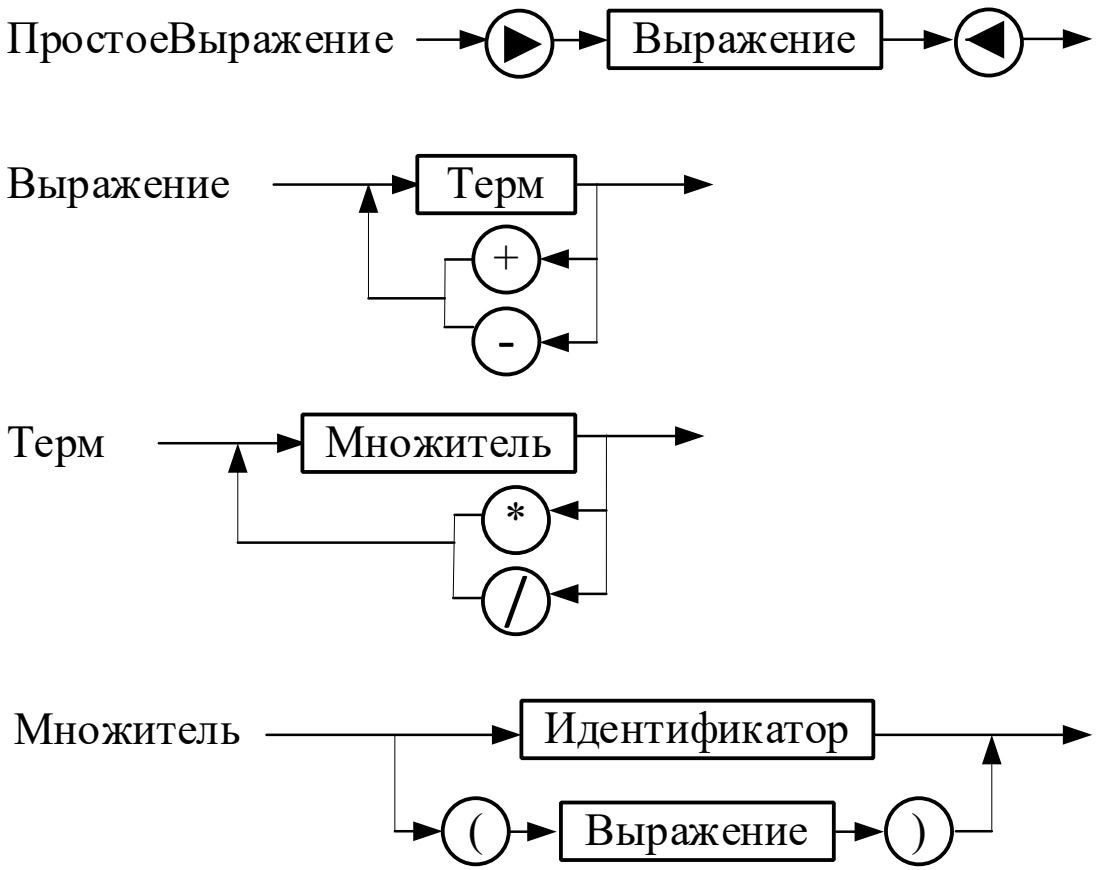
$F \subseteq Q$ – множество заключительных (допускающих) состояний.

Синтаксические диаграммы выражения

Пример. Синтаксический анализатор выражений.

Σ = {<Ид>, +, –, *, /, (,), ◄, ►}.

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



Объединенная синтаксическая диаграмма

ПростоеВыражение

Выражение

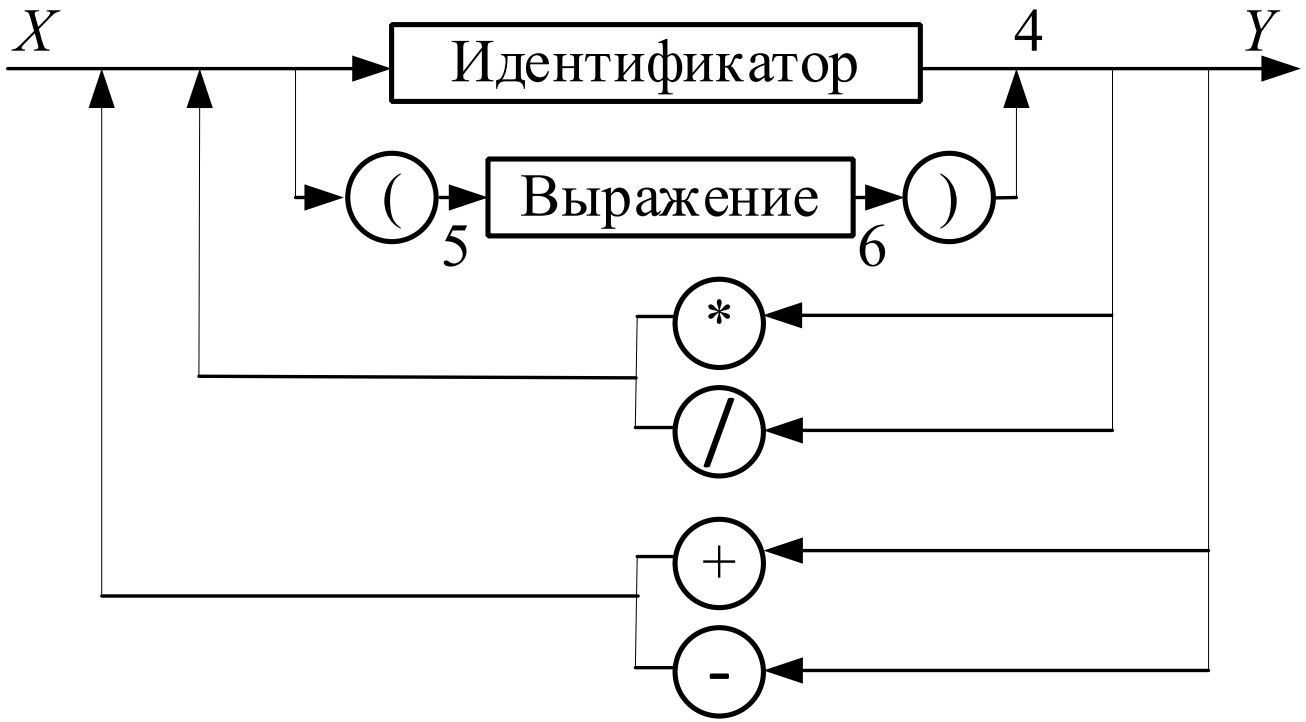


Таблица переходов автомата

	<Ид>	+	-	*	/	(	)	►	◄
1								2	
2						↓ *Y*=3			
3									К

	<Ид>	+	-	*	/	(	)	►	◄
X	4					5			
4		*X*	*X*	*X*	*X*		↑ *Y*		↑ *Y*
5						↓ *Y*=6			
6							4		

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

Место LL(k) и LR(k) грамматик среди грамматик типа 2:

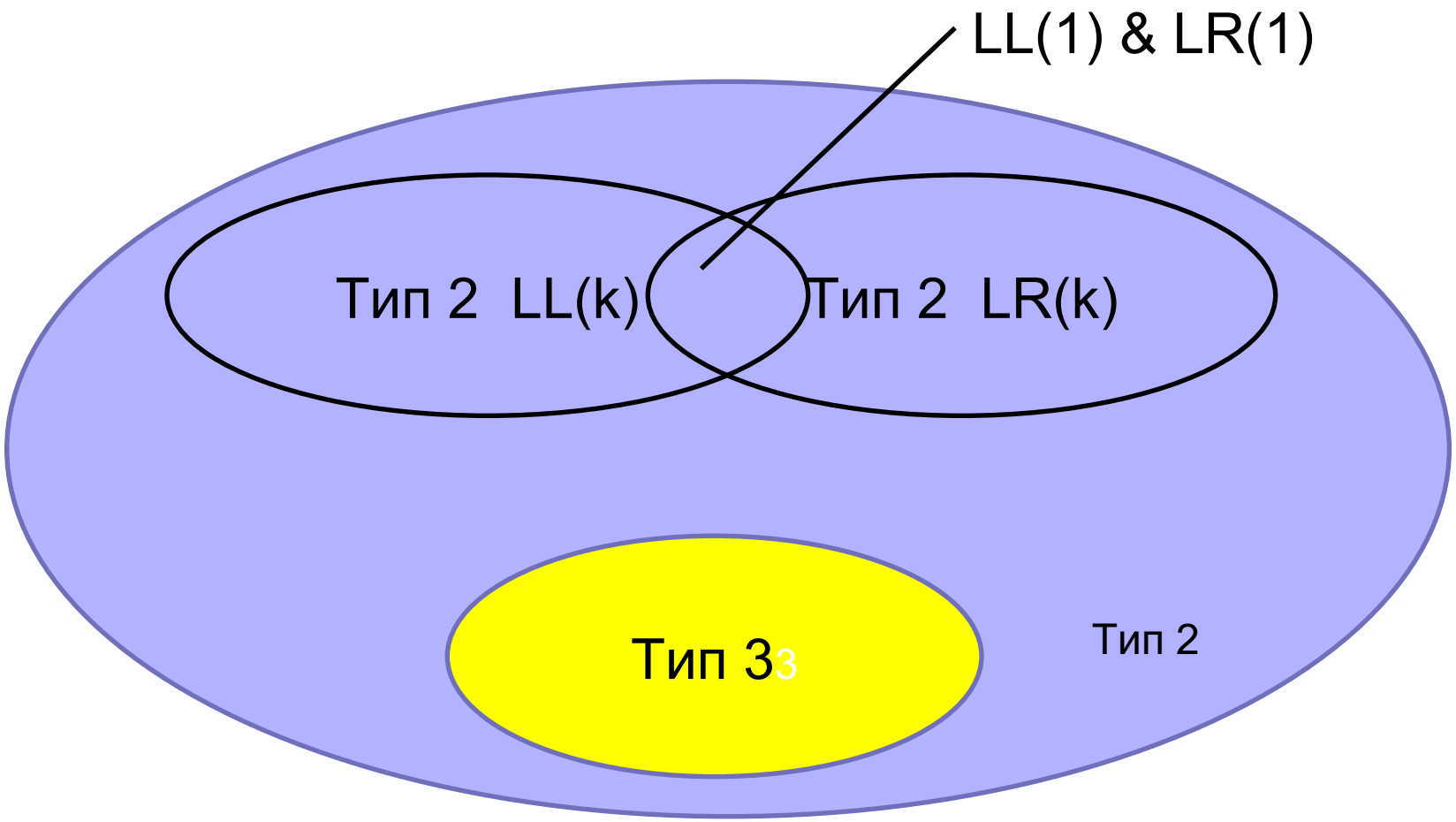
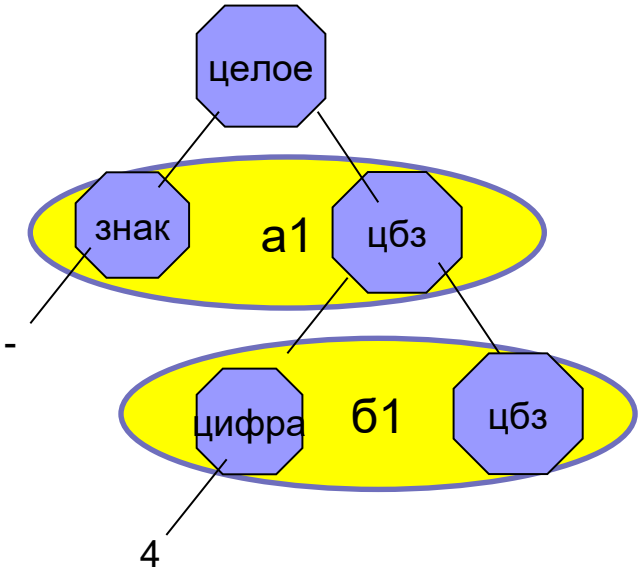


Таблица грамматического разбора левостор. нисход. м.

№ шага	Распо-знано	Распозна-ваемая строка	Строка правил	Действие
1		-45	<Целое>	Подстановка правила а1
2		-45	<Знак><ЦБЗ>	Подстановка правила г1
		-45	+<ЦБЗ>	Ошибка, возврат к шагу 2
3		-45	<Знак><ЦБЗ>	Подстановка правила г2
		-45	- <ЦБЗ>	**Символ распознан**
4	-	45	<ЦБЗ>	Подстановка правила б1
5	-	45	<Цифра><ЦБЗ>	Подстановка правила в1
	-	45	0 <ЦБЗ>	Ошибка, возврат к шагу 5
6	-	45	<Цифра><ЦБЗ>	Подстановка правила в2
	-	45	1 <ЦБЗ>	Ошибка, возврат к шагу 6
7	-	45	<Цифра><ЦБЗ>	Подстановка правила в3
	-	45	2 <ЦБЗ>	Ошибка, возврат к шагу 7
8	-	45	<Цифра><ЦБЗ>	Подстановка правила в4
	-	45	3 <ЦБЗ>	Ошибка, возврат к шагу 8
9	-	45	<Цифра><ЦБЗ>	Подстановка правила в5
	-	45	4 <ЦБЗ>	**Символ распознан**



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:

а) <ПростоеВыр.> ::= ► <Выр>◄

б) <Выр> ::= <Терм> <Слож>

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

г) <Терм> ::= <Множ><Умн>

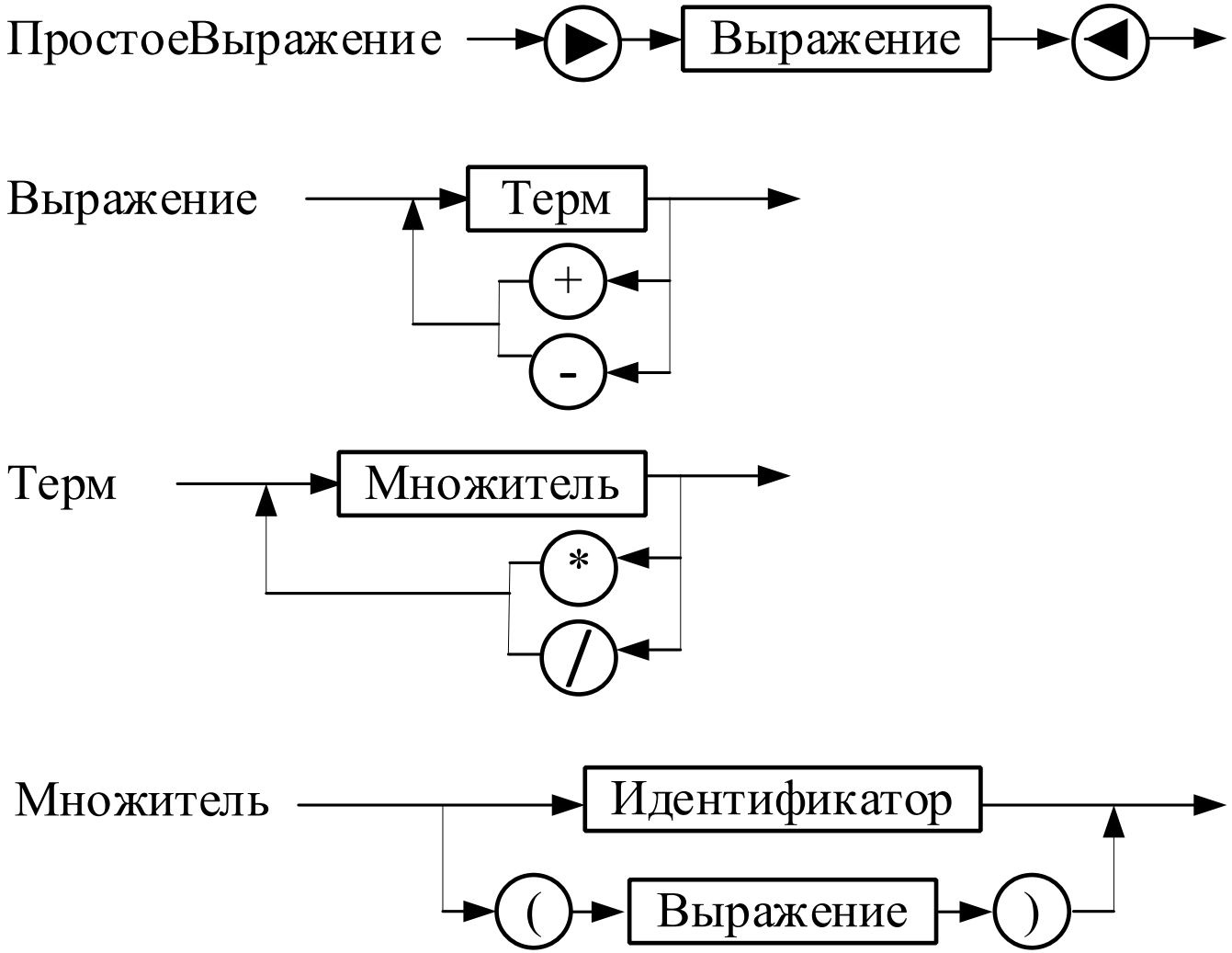
д) <Умн> ::= e | *<Множ><Умн> | / <Множ><Умн>

е) <Множ> ::= <Ид> | (<Выр>)

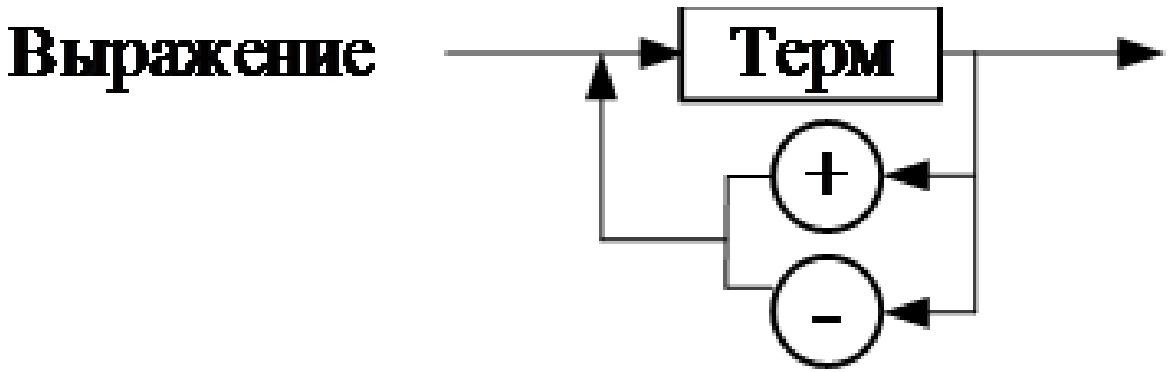
№	Распознаваемая строка	Строка правил	Действие
1	<Ид>*(<Ид>+<Ид>)+<Ид>	<**Выр**>	Подстановка правила а
2	<Ид>*(<Ид>+<Ид>)+<Ид>	**<Терм>**<Слож>	Подстановка правила в
3	<Ид>*(<Ид>+<Ид>)+<Ид>	<**Множ**><Умн><Слож>	Подстановка правила д2
4	<Ид>*(<Ид>+<Ид>)+<Ид>	<Ид><Умн><Слож>	Символ распознан
5	*(<Ид>+<Ид>)+<Ид>	<**Умн**><Слож>	Подстановка правила г1
6	*(<Ид>+<Ид>)+<Ид>	*<Множ><Умн><Слож>	Символ распознан

Метод рекурсивного спуска

Пример. Выражение



Алгоритм распознавателя



Функция Выражение:Boolean:

R:=Терм()

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

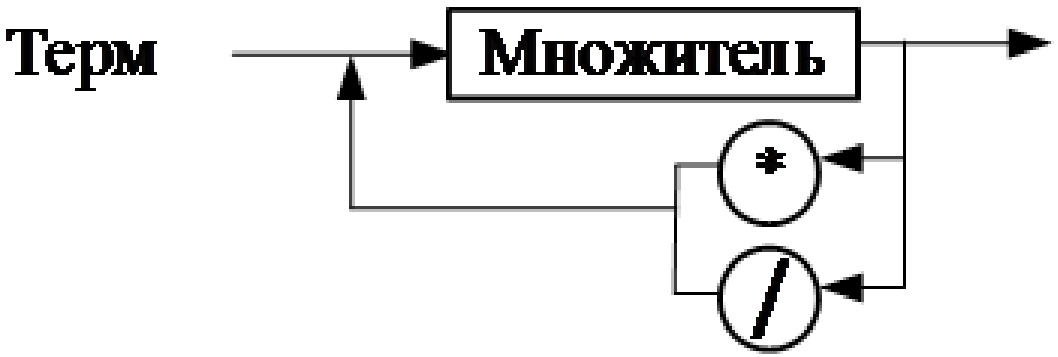
R:=Терм()

Все-цикл

Выражение:= R

Все

Алгоритм распознавателя



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:

Если NextSymbol ≠ ' ►'

то Ошибка() Все-если

R:=Выражение()

Если NextSymbol ≠ ‘◄’

то Ошибка () Все-если

Конец

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:

№ шага	Распознаваемая строка	Основа	
1	-45	-	Свертка по п
2	**<Знак>** 45	<Знак>	Нет правила для свертки. Формирование следующей основы
3	**<Знак> 45**	<Знак> 4	Нет правила для свертки. Формирование следующей основы
4	<Знак> **45**	**4**	Свертка по правилу в5
5	**<Знак>** <Цифра> 5	<Знак>	Нет правила для свертки. Формирование следующей основы
6	**<Знак> <Цифра>** 5	<Знак> <Цифра>	Нет правила для свертки. Формирование следующей основы
7	<Знак> <**Цифра**> 5	<Цифра>	Свертка по правилу б2
8	**<Знак>** <Цбз> 5	<Знак>	Нет правила для свертки. Формирование следующей основы
9	**<Знак> <Цбз>** 5	<Знак>	
10	<Целое> 5	Аксиом	ние

Возвраты возникают из-за неверного выбора основы

Грамматики LR(k) обеспечивают однозначный выбор основы

Грамматики предшествования

Пусть задана некоторая произвольная строка – предложение языка:

....αβ....

Если два символа строки 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> ::= <Выражение>+<Терм>|<Выражение>- <Терм> | <Терм>
<Терм> ::= <Терм>* <Множитель> | <Терм> / <Множитель> | <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 построения синтаксического анализатора

Разработать программу, осуществляющую:

1) лексический анализ:
   a) идентификаторов,
   b) служебных слов,
2) синтаксический анализ:
   a) сравнений (не более одной операции сравнения вида =, <>, >, <, >=, <=),
   b) выражений с операциями +, -, *, / и скобками,
   c) операторов условной передачи управления,
   d) операторов присваивания

   в синтаксисе языка Паскаль.

Например:

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

Алфавит языка. Правила синтаксиса для распознавания лексем

```
if aaaa>vv12 then j:=hhhh
else if h then ffff:=hhh+(ppp+yyy)*k21;
```

Алфавит языка:

V_T = {a..z,A..Z,0..9,if,then,else,:=,+,-,*,/,(,),
=,<>,>,<,>=,<=}

Синтаксис базовых понятий языка для лексического анализа:

<Идентификатор>:: = <Буква>|<Идентификатор><Буква>|
<Идентификатор><Цифра>

<Буква> ::= a|b|c|…|z|A|B|C...|Z

<Цифра> ::= 0|1|2|3|4|5|6|7|8|9

Описание конструкций языка

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

<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>

<Выражение> ::= <Слагаемое>|<Выражение>+<Слагаемое>|
<Выражение>-<Слагаемое>

<Слагаемое> ::= <Множитель>|<Слагаемое>*<Множитель>|
<Слагаемое>/<Множитель>

<Множитель> ::= <Идентификатор> |(<Выражение>)

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 *Обратная польская запись. Алгоритм Бауэра-Замельзона (стековая машина)*

Обратная польская запись или постфиксная запись Яна Лукасевича представляет собой запись математического выражения в виде последовательности 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

Этап 1. 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.

Этап 2. Выполнение польской записи

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

Построение польской записи:

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

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

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.. .

Размещение переменных разных классов

№	Класс памяти C++	В Паскале	Размещение	Видимость	Время жизни	Тип распределения
1	Внешние extern	Глобальные	В основном сегменте данных программы	Программа и подпрограммы, если отсутствует перекрытие имен в подпрограммах	От запуска программы до ее завершения	Статическое
2	Внешние статические extern static	Переменные модуля	В сегменте данных модуля	Для подпрограмм файла (модуля)	От подключения файла (модуля) до его отключения	Статическое
3	Автоматические auto	Локальные	В стеке	Из подпрограммы, в которой объявлены, и вызываемых из нее подпрограмм	От момента вызова подпрограммы до ее завершения	Автоматическое
4	Статические static	-	В основном сегменте данных программы	Из подпрограммы, в которой объявлены, и вызываемых из нее подпрограмм	От запуска программы до ее завершения	Статическое

4.4 Генерация кодов

Генерация кодов – процесс замены распознанных конструкций на соответствующий заранее заготовленный фрагмент.

Пример «заготовки»:

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

Каждый подставляемый фрагмент программно настраивается по данным, находящимся в таблице разбора.

Полученная программа очень далека от оптимальной, поэтому осуществляется ее автоматическая оптимизация еще на уровне таблицы.

Оптимизация кодов

Используются два критерия эффективности результирующей программы :

- **время** выполнения программы;
- **объем памяти**, необходимый для выполнения программы.

В общем случае задача построения оптимального кода программы алгоритмически неразрешима. Основная оптимизация программы должна производится программистом.

Различают две группы методов:

- *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.

Машинно-независимая оптимизация

1. Удаление недостижимого кода

Задача компилятора найти и исключить код, на который не передается управление.

Пример:

```
if (1)
     S1;          ⇒      S1;
else
     S2;
```

Машинно-независимая оптимизация

2. Оптимизация линейных участков программы.

В современных системах программирования профилировщик на основе результатов запуска программы выдаёт информацию о том, на какие её линейные участки приходится основное время выполнения. Для этих фрагментов выполняется:

а) Удаление бесполезных присваиваний.

a = b * c; d = b + c; a = d * c; $\Rightarrow$ ***d = b + c; a = d * c;***

Однако, в следующем примере эта операция уже не бесполезна:

p = & a; a = b * c; d = *p + c; a = d * c;

б) Исключение избыточных вычислений.

d = d + b * c; a = d + b * c; c = d + b * c; $\Rightarrow$

t = b * c; d = d + t; a = d + t; c = a;

в) Свертка объектного кода (выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны).

i = 2 + 1; j = 6 * i + i; $\Rightarrow$ ***i = 3; j = 21;***

Машинно-независимая оптимизация (2)

г) Перестановка операций (для дальнейшей свертки или оптимизации вычислений).

a = 2 * b * 3 * c; ⇒ a = (2 * 3) * (b * c);
a = (b + c) + (d + e); ⇒ a = (b + (c + (d + e)));

д) Арифметические преобразования (на основе алгебраических и логических тождеств).

a = b * c + b * d; ⇒ a = b * (c + d);
a * 1 ⇒ a, a * 0 ⇒ 0, a + 0 ⇒ a .

е) Оптимизация вычисления логических выражений.

a || b || c || d; ⇒ ***a,*** если **a** есть **true**.
Но! ***a || f(b) || g(c)*** не всегда **a** (при **a = true**),
может быть побочный эффект.

Машинно-независимая оптимизация (3)

3. Подстановка кода функции вместо ее вызова в объектный код.

Этот метод, как правило, применим к простым функциям и процедурам, вызываемым непосредственно по адресу, без применения косвенной адресации через таблицы RTTI (Run Time Type Information).

Некоторые компиляторы допускают применять метод только к функциям, содержащим последовательные вычисления без циклов.

Язык C++ позволяет явно указать (inline), для каких функций желательно использовать inline-подстановку.

Машинно-независимая оптимизация (4)

4. Оптимизация циклов.

а) Вынесение инвариантных вычислений из циклов.

```
for (i=1; i<=10; i++)a[i]=b*c*a[i];
⇒
d = b*c;
for (i=1; i<=10; i++) a[i]=d*a[i];
```

б) Замена операций с индуктивными (образующими арифметическую прогрессию) переменными (как правило, умножения на сложение).

```
for (i=1; i<=N; i++) a[i]=i*10;
⇒
t = 10; i = 1;
while (i<=N) { a[i]=t;  t=t+10;  i++;}

s = 10;
for (i=1; i<=N; i++) { r=r+f(s); s=s+10; }
⇒
s=10; m=N*10;
while (s <= m) {r= r+f(s); s=s+10; }
```

(избавились от одной индуктивной переменной).

Машинно-независимая оптимизация (5)

в) **Слияние циклов**.

```
for (i=1; i<=N; i++)
    for (j= 1; j<= M; j++) a[i][j] = 0;
```

⇒

```
k = N * M;
for (i=1; i<=k; i++)
      a[i] = 0;
```

(только в объектном коде!)

г) **Развертывание циклов** (можно выполнить для циклов, кратность выполнения которых известна на этапе компиляции).

```
for (i=1; i<=3; i++) a[i]=i;
```

⇒

```
a [1] = 1;
a [2] = 2;
a [3] = 3;
```

Машинно-зависимая оптимизация

Машинно-зависимые преобразования результирующей объектной программы зависят от архитектуры вычислительной системы, на которой будет выполняться результирующая программа. При этом может учитываться объем кэш-памяти, методы организации работы процессора ….

Эти преобразования, как правило, являются "ноу-хау", и именно они позволяют существенно повысить эффективность результирующего кода.

.

1. **Распределение регистров процессора.**

Использование регистров общего назначения и специальных регистров (аккумулятор, счетчик цикла, базовый указатель) для хранения значения операндов и результатов вычислений позволяет увеличить быстродействие программы.

Доступных регистров всегда ограниченное количество, поэтому перед компилятором встает вопрос их оптимального распределения и использования при выполнении вычислений.

Машинно-зависимая оптимизация

2. Оптимизация передачи параметров в процедуры и функции.

Обычно параметры процедур и функций передаются через стек. При этом всякий раз при вызове процедуры или функции компилятор создает объектный код для размещения ее фактических параметров в стеке, а при выходе из нее - код для освобождения соответствующей памяти.

Можно уменьшить код и время выполнения результирующей программы за счет оптимизации передачи параметров в процедуру или функцию, передавая их **через регистры** процессора.

Реализация данного оптимизирующего преобразования зависит от количества доступных регистров процессора в целевой вычислительной системе и от используемого компилятором алгоритма распределения регистров.

Недостатки метода:

- оптимизированные таким образом процедуры и функции не могут быть использованы в качестве **библиотечных**, т.к. методы передачи параметров через регистры не стандартизованы и зависят от реализации компилятора.
- этот метод не может быть использован, если где-либо в функции требуется выполнить **операции с адресами** параметров.

Языки Си и C++ позволяют явно указать (**register**), какие параметры и локальные переменные желательно разместить в регистрах.

Машинно-зависимая оптимизация

3. Оптимизация кода для процессоров, допускающих распараллеливание вычислений.

При возможности параллельного выполнения нескольких операций компилятор должен порождать объектный код таким образом, чтобы в нем было максимально возможное количество соседних операций, все операнды которых не зависят друг от друга.

Для этого надо найти оптимальный порядок выполнения операций для каждого оператора (переставить их).

$a + b + c + d + e + f; \Rightarrow$

для одного потока обработки данных: $((((a + b) + c) + d) + e) + f;$

для двух потоков обработки данных: $((a + b) + c) + ((d + e) + f);$

для трех потоков обработки данных: $(a + b) + (c + d) + (e + f);$