

Министерство образования и науки Российской Федерации  
ФГАО УВО “Севастопольский государственный университет”



## МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению контрольных работ по дисциплине  
“Теоретические основы построения компиляторов”  
для студентов заочной формы обучения основного профиля  
09.03.02 – “Информационные системы и технологии”



Севастополь  
2015

УДК 004.4

Методические указания к выполнению контрольных работ по дисциплине “Теоретические основы построения компиляторов” для студентов заочной формы обучения основного профиля 09.03.02 – “Информационные системы и технологии” [Текст] / Разраб. В.Ю. Карлусов, С.А. Кузнецов. - Севастополь: Изд-во СевГУ, 2015. – 36 с.

***Цель методических указаний:*** Обеспечение студента заочной формы обучения дидактическим материалом для качественного выполнения контрольных работ; по разделам системного программирования: "формальные языки и грамматики", "элементы теории конечных автоматов", "отношения предшествования" в приложении к задачам синтаксического анализа программ на алгоритмических языках.

Методические указания рассмотрены и утверждены на заседании кафедры Информационных систем,  
протокол № 01 от 03 февраля 2015 г.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

***Рецензент*** Кожаяев Е.А., кандидат техн. наук, доцент кафедры кибернетики и вычислительной техники.

## Общие положения

Студент за время обучения выполняет две контрольные работы по дисциплине: №1 – «Построение лексического анализатора», №2 – «Построение синтаксического анализатора».

Вариант является сквозным для обеих контрольных работ и выбирается, используя две последние цифры зачётной книжки, по таблицам, содержащимся в приложении А.

В общем случае вариант представляется тремя кодами в таблице А.1, разделенными точкой, например 13.7.9.

Первый код относится к описанию идентификаторов языка учебной программы, содержащемуся в таблице А.2. Следующая цифра относится к типу констант, который описывается в таблице А.3. Последнее число позволяет определить по таблице А.4 множество разделителей, служебных слов программы и их компоновку в операторах языка учебной программы.

Выходная информация лексического анализатора впоследствии используется в качестве входных данных для анализатора синтаксиса.

**К оформлению контрольных работ предъявляются следующие требования.** Для контрольной работы №1 в отчёт включаются описания в виде регулярных выражений отдельных лексем и всего лексического анализатора, управляющие таблицы детерминированного конечного автомата до и после минимизации, список диагностических сообщений. Работу лексического анализатора студент демонстрирует на сессии.

Для контрольной работы № 2 в отчёте должен быть отражён ход построения формальной грамматики, выполнение исследований её на свойства  $LL(k)$  и наличие отношений простого предшествования, и, при необходимости, эквивалентные преобразования грамматики к соответствующему классу. Прилагаются текст программы нисходящего рекурсивного распознавателя, список диагностических сообщений, разработанных для обоих типов распознавателей.

Работу синтаксического анализатора студент демонстрирует на сессии.

## Контрольная работа № 1 “ПОСТРОЕНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА”

### 1. Краткие теоретические сведения

Лексический анализатор (сканер) решает задачу синтаксического анализа на уровне лексем.

Под лексемой понимается некоторый класс эквивалентности на множестве (словаре) терминальных символов формальной грамматики, порождающей операторы языка программирования.

В задачи лексического анализатора входят: а) выделение лексем (атомов, терминальных символов) языка из потока литер, проверка правильности их написания; б) замена цепочки литер, соответствующей опознанной лексеме, определенным внутренним кодом, так называемым дескриптором; в) ведение таблиц соответствия объектов программы системе кодирования: таблиц констант, таблиц имен переменных, таблиц размещения лексем по строкам; г) устранение пробелов, комментариев и аналогичных элементов визуального восприятия программы, и представление исходной программы последовательностью дескрипторов.

На контрольную работу выносятся пункты а), б), и г) из перечня задач.

Под дескриптором понимают пару вида (код\_класса\_лексемы, ссылка\_на\_значение).

В языке программирования традиционно присутствуют следующие классы лексем: имена переменных (идентификаторы), константы, знаки операций, знаки пунктуации (скобки, разделители списка параметров, разделители операторов).

Иногда, с целью упрощения процедур синтаксического анализа, классы знаков операций и знаков пунктуации сводят до одного символа в каждом классе.

Лексический анализ выполняется с использованием детерминированных конечных автоматов. Поэтому для построения сканера необходимо разработать конечный автомат, выполняющий классификацию цепочек литер, и в заключительных состояниях помещающий необходимый дескриптор в файл или массив выходных данных.

Процедура построения сканера включает следующие шаги:

1. Определение лексем языка программирования.
2. Назначение кодов лексемам.
3. Проектирование конечного автомата, на базе которого будет осуществляться анализ.
4. Составление перечня диагностических сообщений об ошибках в тех случаях, когда функция перехода конечного автомата не определена.

5. Определение алгоритмов и процедур, связанных с обработкой информации в пределах множества состояний конечного автомата.
6. Разработка тестовых последовательностей.
7. Программирование и отладка сканера.

## 2. Пример выполнения контрольной работы

### 2.1. Пусть исходные данные имеют вид, представленный ниже

Служебные слова:	STA – начало программы. STO – окончание.
Описание идентификатора:	начинается буквой латинского алфавита, а заканчивается буквой E.
Однолитерные разделители:	+, -, *, /. .
Двулитерные разделители:	: =.
Константа:	восьмеричная дробь, например – 617.214
Фрагмент программы для анализа:	STA $xE := yE + 1.1$ $y123E := xE * yE - 2.257$ STO

### 2.2 Определение лексем и назначение им кодов

Коды лексемам назначим, руководствуясь соображениями об их уникальности и удобстве эффективной обработки, поэтому класс однолитерных разделителей имеет, например, в коде префикс 5.

STA	100
STO	200
идентификатор, <iden>	300
константа, <data>	400
+	501
-	502
*	503
/	504
:=	600

## 2.3 Построение минимального детерминированного конечного автомата

Точное описание конечного автомата удастся получить на основании цепочек литер с использованием регулярных выражений. Правила конструирования регулярных выражений изложены в [9, 10, 12].

Общая структура регулярного выражения примерно такова:

$$C_{s_1}L_1 \cup C_{s_2}L_1 \cup \langle \text{iden} \rangle L_1 \cup \langle \text{data} \rangle L_2 \cup d_1 \cup d_2 \cup \dots \cup d_j \cup \dots \cup d_n \cup L_3.$$

В записи обозначено:

$C_{s_i}$  –  $i$ -тое служебное слово.

$L_1$  – лексема, состоящая в том, что текущая литера не буква или не цифра.

$L_2$  – текущая литера не восьмеричная цифра.

$d_j$  – разделители  $j$ -того типа.

$L_3$  – литера, не принадлежащая алфавиту конечного автомата.

Служебные слова являются самоопределяющимися цепочками, составленными из литер  $\{S, T, A, O\}$  – STA, STO.

Эскизно идентификатор (переменная) будет выглядеть так

$$B \{ B \cup C \} E,$$

где  $B$  – любая латинская буква,  $C$  – любая десятичная цифра.

Восьмеричная дробная константа описывается следующим образом:

$$\{u\} u \bullet \{u\} u,$$

где  $u$  – восьмеричная цифра из диапазона  $0 \div 7$ .

С учетом того, что конечный автомат должен отличать отдельные буквы и цифры на фоне других, имеем:

1. Множество букв  $B = \{S, T, A, O, E, \delta\}$ .

$\delta$  – буква латинского алфавита, не совпадающая по начертанию с S, T, A, O, E.

2. Множество цифр  $C = \{8, 9, u\}$

3. Множество «не буква, не цифра»  $L_1 = \{+, -, *, /, :=, \bullet\} \cup L_3$

4. Множество «не восьмеричная цифра»  $L_2 = B \cup \{8, 9\} \cup L_1$

Окончательно получаем дизъюнктивные члены выражения, описывающие конечный автомат:

1.  $STAL_1$  – первое служебное слово.

2.  $STOL_1$  – второе служебное слово.

3.  $(S \cup T \cup A \cup O \cup E \cup \delta) \{S \cup T \cup A \cup O \cup E \cup \delta \cup 8 \cup 9 \cup u\} EL_1$  – переменная.

4.  $\{u\} u \bullet \{u\} u L_2$  – константа.

5.  $+ \cup - \cup * \cup / \cup := \cup L_3$  – однолитерные и двулитерные разделители.

Регулярное выражение подвергается минимизации по алгоритму, изложенному в [8].

Процесс разметки и минимизации представлен на рисунке 1.1. На фрагменте 1.1, а, показано выражение до минимизации, на 1.1, б, – окончательно минимизированное выражение.

В результирующем выражении видно, что по цепочкам литер, в которых начало идентификатора полностью или частично совпадает со служебным сло-

вом, автомат является недетерминированным. Это образы переходов  $\delta(1, T) = \{2, 13\}$ ,  $\delta(\{2, 13\}, A) = \{3, 14\}$  и т.д.

Для приведения недетерминированного конечного автомата к детерминированному, можно воспользоваться классическими алгоритмами приведения [7 – 10], но можно воспользоваться описанием КА в виде графа и с его помощью привести таблицу переходов ДКА. Фрагмент графа конечного автомата представлен на рисунке 1.2, а в таблице 1.1 – представлена функция переходов конечного автомата.

Фрагмент графа конечного автомата, отображенный на рисунке 1.2, показан в столбцах таблицы 1.1 с номерами 1 и 2 значениями, заключенными в скобки. Прочие элементы указанных столбцов записываются согласно разметке части регулярного выражения, относящегося к описанию идентификатора регулярного выражения, заключенного в фигурные скобки (смотрите рисунок 1.1,б, основные места с 12-го по 20-е).

Приступим к минимизации таблицы, описывающий конечный автомат. Приемы минимизации изложены в [4, 9, 11], согласно им, под минимизацию подпадают столбцы таблицы, соответствующие состояниям конечного автомата, не различимым по входной строке и по выходному сигналу. В качестве выходных сигналов выберем следующие: сигнал рабочего состояния для всех состояний, кроме заключительных, соответствующих опознанным лексемам, это 4, 6, 21, 25 – 28, 31, 32. В каждом из этих состояний генерируется свой уникальный сигнал. Последние состояния можно удалить из таблицы как неинформативные, хотя функции перехода КА в эти состояния оставлены.

Множества неразличимых состояний, минимизируемые состояния, суть  $\{7 – 15, 17 – 20\}$ . Конечные состояния  $\{4, 6, 21, 25 – 28, 31, 32\}$  тоже устраним из таблицы переходов, а соответствующую позицию функции  $\delta(q_r, a_r)$  определим используя коды, присвоенные этим лексемам на этапе кодирования.

Для осуществления корректной минимизации, в таблице 1.2 приводится система перенумерации состояний для таблицы минимального автомата.

Для случаев, когда образ функции переходов КА пуст ( см. исходную таблицу 1.1), введём коды ошибок, которые в ходе работы сканера помогут генерировать осмысленные диагностические сообщения.

1. Некорректный фрагмент цепочки. Состояния 0, строки соответствующие литерам  $\{8, 9, \bullet, =\}$ , код 801.

2. Ошибка в служебном слове. Состояния  $\{1, 2\}$ , строки, соответствующие  $L_1, L_3$ , код 802.

3. Ошибка в написании имени переменной. Состояния  $\{7 – 15, 17 – 20\}$ , строки соответствующие  $L_1, L_2$ , код 803.

4. Ошибочная константа. Состояние 22, строки, соответствующие множеству  $L_2 \setminus \text{”}\bullet\text{”}$ . Состояние 23, все строки, кроме «ц», код 804.

5. Ошибка в операторе присваивания. Состояние 30, все строки, кроме «=», код 805.





	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32							
S	1				Служебное слово STA	12	Служебное слово STO	12	12	12	12	12	12	12	12	12	12	12	12	12	12	Идентификатор			25	Восьмеричная константа	Знак +	Знак -	Знак *	Знак /										
T	7	(2)	13	13		13		13	13	13	13	13	13	13	13	13	13	13	13	13	13		13								25									
A	8	14	(3)	14		14		14	14	14	14	14	14	14	14	14	14	14	14	14	14		14								25									
O	9	16	(5)	16		15		16	16	16	16	16	16	16	16	16	16	16	16	16	16		16								25									
E	10	15	15	15		16		15	15	15	15	15	15	15	15	15	15	15	15	15	15		15								25									
δ	11	17	17	17		17		17	17	17	17	17	17	17	17	17	17	17	17	17	17		17								25									
8		18	18	18		18		18	18	18	18	18	18	18	18	18	18	18	18	18	18		18								25									
9		19	19	19		19		19	19	19	19	19	19	19	19	19	19	19	19	19	19		19								25									
ц	22	20	20	20		20		20										21													22	24	24							
+	26			4		6												21														25								
-	27			4	6											21								25																
*	28			4	6											21								25																
/	29			4	6											21								25																
:	30			4	6											21								25																
·				4	6											21							23		25															
=				4	6											21								25							31									
L3	32			4	6											21								25																

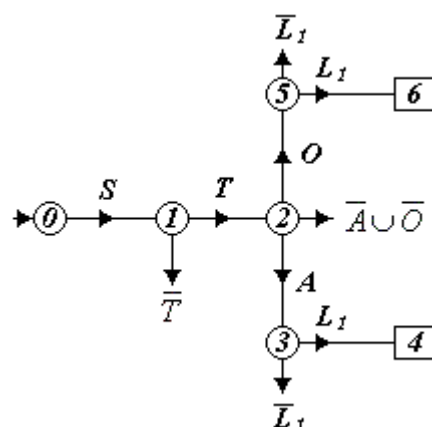


Рисунок 1.2 – Фрагмент КА, относящийся к разбору служебных слов

Функция переходов минимального детерминированного конечного автомата приводится в таблице 1.3.

В алфавит входных символов конечного автомата добавлен пробел, пополняющий множество  $L_1$  (не буква, не цифра) для упрощения подготовки тестовых данных с позиции их наглядности (читабельности).

Таблица 1.2 Соответствие старых и новых состояний МДКА

Множество старых состояний	Новые состояния
0	0
1	1
2	2
3	3
4	Исключено, код 100
5	4
6	Исключено, код 200
7-14,16-20	5
15	6
21	Исключено, код 300
22	7
23	8
24	9
25	Исключено, код 400
26	Исключено, код 501
27	Исключено, код 502
28	Исключено, код 503
29	Исключено, код 504
30	10
31	Исключено, код 600
32	Исключено, код 700

## 2.4 Методология разработки программы сканера

В общем случае алгоритм работы программы моделирования автомата может выглядеть таким образом:

организация цикла чтения из файла по одной литере до достижения его конца. Номер состояния КА предварительно должен быть положен 0, файл, в который будут записаны дескрипторы, открыт. В теле цикла выполняются следующие действия:

1. По текущей литере определяется номер строки таблицы функции переходов.

2. Пара (номер строки, номер состояния) однозначно определяет значение функции перехода.

3. Код значения функции перехода анализируется на принадлежность к :  
а) кодам рабочих состояний; б) кодам заключительных состояний; в) кодам ошибок.

Для вариантов б) и в) автомат устанавливается в начальное состояние. В зависимости от результатов анализа формируется выходной файл (случай б), диагностическое сообщение (случай в) или никаких действий не выполняется (случай а).

Таблица 1.3 – Переходы минимального ДКА

№		0	1	2	3	4	5	6	7	8	9	10	
0	S	1	5	5	5	5	5	5	804	804	400	805	L <sub>2</sub>
1	T	5	2	5	5	5	5	5	804	804	400	805	
2	A	5	5	3	5	5	5	5	804	804	400	805	
3	O	5	5	4	5	5	5	5	804	804	400	805	
4	E	6	6	6	6	6	6	6	804	804	400	805	
5	δ	5	5	5	5	5	5	5	804	804	400	805	
6	8	801	5	5	5	5	5	5	804	804	400	805	
7	9	801	5	5	5	5	5	5	804	804	400	805	
8	Ц	7	5	5	5	5	5	5	7	9	9	805	L <sub>1</sub> , L <sub>2</sub>
9	+	501	802	802	100	200	803	300	804	804	400	805	
10	-	502	802	802	100	200	803	300	804	804	400	805	
11	*	503	802	802	100	200	803	300	804	804	400	805	
12	/	504	802	802	100	200	803	300	804	804	400	805	
13	:	10	802	802	100	200	803	300	804	804	400	805	
14	•	801	802	802	100	200	803	300	8	804	400	805	
15	=	801	802	802	100	200	803	300	804	804	400	600	
16		0	802	802	100	200	803	300	804	804	400	805	
17	L3	700	802	802	100	200	803	300	804	804	400	805	

Обратим внимание, что при опознавании сканером служебного слова, переменной или константы номер состояния КА полагается равным начальному, а **чтение очередной литеры из входного файла не выполняется!**

Для эффективного осуществления кодирования алгоритма, предполагается использовать операторы `switch`, `do...while`, `if` и встроенные функции `isalpha()`, `isdigit()` для экспресс – определения принадлежности литеры и `toupper()` для принудительной установки регистра букв в положение прописной буквы.

Фрагмент программы представлен ниже.

```
N_sost = 0;
if (( c = getc( )) != EOF) do {
    if (isalpha(c)) /* Это буква? */
        switch(toupper(c) {
            case "S": N_str=0;break;
            ...
            case "E": N_str=4;break;
            default: N_str=5; } /* δ */
    else if (isdigit(c)) /* Это цифра? */
        switch (c) {
            case "8": N_str=6;break;
            case "9": N_str=7;break;
            default: N_str=8; } /* 8-я цифра */
    else switch (c) {
        /* Это ни буква, ни цифра ? */
        case "+": N_str=9;break;
        ...
        case " ": N_str=16;break;
        default: N_str=17; }
    N_sost=upr_table[N_str][N_sost];
    /* Получение состояния перехода */
    if ((N_sost>99)&&(N_sost<500)) {
        /* Анализ кода состояния */
        fprintf(Prog_code,"%d",N_sost);N_sost=0; }
    else if (N_sost>500)&&(N_sost<800) {
        fprintf(Prog_code,"%d",N_sost);
        N_sost=0;c=getc( );}
    else if (N_sost>800) {
        switch(N_sost) {
            case 801:
                printf("\n Неправильное начало");break;
            case 805:
                printf("\n Ожидалось равно");break;
            default:
```

```

        printf("\n Неизвестная входная литера");
    }
    N_sost=0; c= getc( ); }
    else c= getc( );
    } while (c!=EOF);

```

В приведенном фрагменте обозначено: `N_sost` – переменная, содержащая текущий номер состояния конечного автомата или код ошибки или лексемы; `c` – переменная, содержащая текущую литеру на входе конечного автомата; `Prog_code` – управляющая переменная файла, принимающего дескрипторы лексем.

Отметим, что в учебной программе отсутствуют компоненты для формирования таблиц имен и констант, поиска в этих таблицах, а сам дескриптор представлен только кодом класса лексем без ссылки на значение.

Разработанные в ходе проектирования тестовые последовательности литер приведены в таблице 1.4.

Таблица 1.4 – Правильные и ошибочные цепочки

<i>Цепочка литер</i>	<i>Результат обработки</i>
STA	Допущено, код 100
+	Допущено, код 501
=	Не допущено, код 801
1.178	Допущено 1.17, код 400. Не допущено 8, код 801
:=	Допущено, код 600
E+STE “пробел”	Допущено, коды 300, 501, 300
SST-	Не допущено SST, код 803. Допущен – , код 502
STO*ST:=	Допущено, код 200, 503, ошибка 802, допущен код 600

## Контрольная работа № 2 “ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА”

### 1.Краткие теоретические сведения

В процессе синтаксического анализа определяется общая структура программы, что включает проверку порядка следования компоновки, размещения символов в программе [3 – 5, 7, 8, 10]. Предполагается, что в качестве входной информации синтаксического анализатора служит последовательность кодов лексем, а результатом является древовидное представление программы, называемое синтаксическим деревом. Построение синтаксического дерева может

выполняться как в явном, так и в неявном виде в ходе выполнения алгоритма синтаксического анализа. Также существуют различные способы, регламентирующие порядок его построения [4, 8]. Различают построение синтаксического дерева от вершины, в качестве которой принимается аксиома грамматики, к продолжению языка. В этом случае употребляют термины: нисходящий анализ, нисходящий распознаватель etc. Противоположный, то есть в направлении от разбираемого предложения к вершине, анализ осуществляется алгоритмом восходящего разбора. По характеру работы синтаксические распознаватели могут использовать прямые (эвристические) алгоритмы и синтаксические управляемые алгоритмы, ориентированные на те или иные свойства формальных грамматик. Использование последних позволяет прогнозировать ход разбора, тем самым повышая эффективность работы компилятора, в целом, и распознавателя, в частности.

В контрольной работе №2 выполняется проектирование нисходящего распознавателя, работа которого основана на свойствах  $LL(k)$  грамматик по методу синтаксических функций, и восходящего распознавателя, основанного на отношениях простого предшествования, называемых также отношениями Вирта-Вебера [2]. Работа выполняется следующим образом.

1. На основании варианта задания (фрагмента программы) разрабатывается формальная грамматика;
2. Продукции грамматики анализируются на свойство  $LL(1)$ , а , по необходимости, исходная грамматика преобразовывается к этому виду;
3. Методом синтаксических функций строится распознаватель рекурсивного спуска;
4. Грамматика, разработанная в пункте 1, исследуется на существование отношений предшествования, а если возникает потребность, то преобразуется к такой форме;
5. Разрабатывается программа, осуществляющая восходящий синтаксический анализ, управляемый отношениями предшествования.

## 2. Построение нисходящего распознавателя

### 2.1. Построение $LL(k)$ -грамматики

Построение грамматики, строго говоря, процесс нетривиальный, хотя и поддающийся, в известной мере, формализации. Схематично “алгоритм” написания формальной грамматики может быть отображен последовательностью операций: описание структуры программы  $\Rightarrow$  описание компонентов структуры  $\Rightarrow$  описание отдельных видов операторов  $\Rightarrow$  описание компонентов операторов.

Примеры полного описания синтаксиса языков программирования приведены в [1, 2]. Объект, который нам предстоит описать, как видно из примера, состоит из служебного слова STA, за которым следует последовательность операторов, завершаемая служебным словом STO. В нотации Бэкуса-Наура это может выглядеть так:

$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{операторы} \rangle \text{STO}.$  (1)

В свою очередь, последовательность операторов может состоять из одного оператора, то есть

$\langle \text{операторы} \rangle ::= \langle \text{оператор} \rangle,$  (2)

либо из оператора, за которым следует последовательность операторов

$\langle \text{операторы} \rangle ::= \langle \text{оператор} \rangle \langle \text{операторы} \rangle.$  (3)

Совершенно аналогично необходимо описать конструкцию оператора, которая представляет собой последовательность, состоящую из имени переменной, знака присваивания и выражения

$\langle \text{оператор} \rangle ::= \langle \text{idен} \rangle := \langle \text{выражение} \rangle.$  (4)

Выражение может быть одиночным операндом

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle,$  (5)

или представлять сложную конструкцию из операндов, разделенных последовательностью действий, то есть

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \langle \text{знак} \rangle \langle \text{выражение} \rangle.$  (6)

Теперь остается определить сущность операнда, который может быть либо именем переменной (идентификатором) либо константой:

$\langle \text{операнд} \rangle ::= \langle \text{idен} \rangle,$  (7)

$\langle \text{операнд} \rangle ::= \langle \text{data} \rangle,$  (8)

и знаков операций.

$\langle \text{знак} \rangle ::= +, \langle \text{знак} \rangle ::= -, \langle \text{знак} \rangle ::= *, \langle \text{знак} \rangle ::= /. \quad (9)-(12)$

Окончательно грамматика будет выглядеть следующим образом:

$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{операторы} \rangle \text{STO},$

$\langle \text{операторы} \rangle ::= \langle \text{оператор} \rangle | \langle \text{оператор} \rangle \langle \text{операторы} \rangle,$

$\langle \text{оператор} \rangle ::= \langle \text{idен} \rangle := \langle \text{выражение} \rangle,$

$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle | \langle \text{операнд} \rangle \langle \text{знак} \rangle \langle \text{выражение} \rangle,$

$\langle \text{операнд} \rangle ::= \langle \text{idен} \rangle | \langle \text{data} \rangle,$

$\langle \text{знак} \rangle ::= + | - | * | /.$

Понятия грамматики, такие как идентификатор ( $\langle \text{idен} \rangle$ ) и константа ( $\langle \text{data} \rangle$ ) обычно описываются в форме Бэкуса-Наура, хотя считаются условно терминальными символами, так как их опознание и контроль правильности написания распознаются на этапе лексического анализа. Мы же это описание опустим, как несущественное при решении нашей задачи.

Элементами разработанной грамматики являются: 1) словарь нетерминалов  $V_n = \{\text{фрагмент}, \text{операторы}, \text{оператор}, \text{выражение}, \text{операнд}, \text{знак}\}$ ; 2) словарь терминалов  $V_t = \{\text{STA}, \text{STO}, +, -, *, /, :=, \langle \text{idен} \rangle, \langle \text{data} \rangle\}$ ; 3) аксиома грамматики – нетерминальный символ  $\langle \text{фрагмент} \rangle$ ; множество продукций – правила подстановки (1)-(12).

Для контроля корректности формальной грамматики, то есть релевантности продукций и непустоты языка, построим синтаксическое дерево, приведенное на рисунке 2.1, хотя исследование этих свойств проводят намного скрупулёзнее [2, 4, 12].

Исследуем грамматику на принадлежность к классу  $LL(k)$ . Обычно процедуру начинают с индекса  $k=1$ . Для того, чтобы грамматика была  $LL(k)$ , необходимо, чтобы цепочки терминальных символов длины  $k$ , получающиеся в ходе вывода с использованием продукций грамматики, для продукций с одинаковыми **левыми** частями, **не пересекались**. Собственно, процедура исследований изложена в [3, 11], проиллюстрирована в [9], результаты исследования грамматики помещены в таблицу 2.1. Из нее следует, что построенная грамматика не является  $LL(1)$  грамматикой. Увеличение индекса  $k$  свыше одного не является желательным, так как вызывает необходимость просмотра контекста анализируемой программы, что ведет к снижению производительности (скорости анализа) распознавателя.

Таблица 2.1 – Исследование  $LL(1)$  – свойств формальной грамматики

№	Продукции грамматики	Цепочки для $k=1$
1	$\langle \text{фрагмент} \rangle \rightarrow \text{STA} \langle \text{операторы} \rangle \text{STO}$	$\{\text{STA}\}$
2	$\langle \text{операторы} \rangle \rightarrow \langle \text{оператор} \rangle$	$\{\langle \text{iden} \rangle\}$
3	$\langle \text{операторы} \rangle \rightarrow \langle \text{оператор} \rangle \langle \text{операторы} \rangle$	$\{\langle \text{iden} \rangle\}$
4	$\langle \text{оператор} \rangle \rightarrow \langle \text{iden} \rangle := \langle \text{выражение} \rangle$	$\{\langle \text{iden} \rangle\}$
5	$\langle \text{выражение} \rangle \rightarrow \langle \text{операнд} \rangle$	$\{\langle \text{iden} \rangle, \langle \text{data} \rangle\}$
6	$\langle \text{выражение} \rangle \rightarrow \langle \text{операнд} \rangle \langle \text{знак} \rangle \langle \text{выражение} \rangle$	$\{\langle \text{iden} \rangle, \langle \text{data} \rangle\}$
7	$\langle \text{операнд} \rangle \rightarrow \langle \text{iden} \rangle$	$\{\langle \text{iden} \rangle\}$
8	$\langle \text{операнд} \rangle \rightarrow \langle \text{data} \rangle$	$\{\langle \text{data} \rangle\}$
9	$\langle \text{знак} \rangle \rightarrow +$	$\{+\}$
10	$\langle \text{знак} \rangle \rightarrow -$	$\{-\}$
11	$\langle \text{знак} \rangle \rightarrow *$	$\{*\}$
12	$\langle \text{знак} \rangle \rightarrow /$	$\{/ \}$

Свойство  $LL(1)$  нарушается для правил (2) и (3), описывающих нетерминальный символ  $\langle \text{операторы} \rangle$ , и правил (5) и (6) для нетерминала  $\langle \text{выражение} \rangle$ . Для приведения грамматики к классу  $LL(1)$  используется стандартный прием, называемый факторизацией [7]. Его сущность заключается в применении метасимволов повторения к правилам, содержащим правую рекурсию. Использование этого подхода приведет к следующему:

$$\langle \text{операторы} \rangle ::= \langle \text{оператор} \rangle \{ \langle \text{оператор} \rangle \}_0^\infty \quad (13)$$

$$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{знак} \rangle \langle \text{операнд} \rangle \}_0^\infty \quad (14)$$

На этапе программной реализации неизбежно возникнут задачи организации цикла, его поддержки и завершения, которые будут в основе своей содержать анализ контекста на глубину, равную 2, что заведомо снизит производительность алгоритма. Хотя факторизация правил грамматики выступает



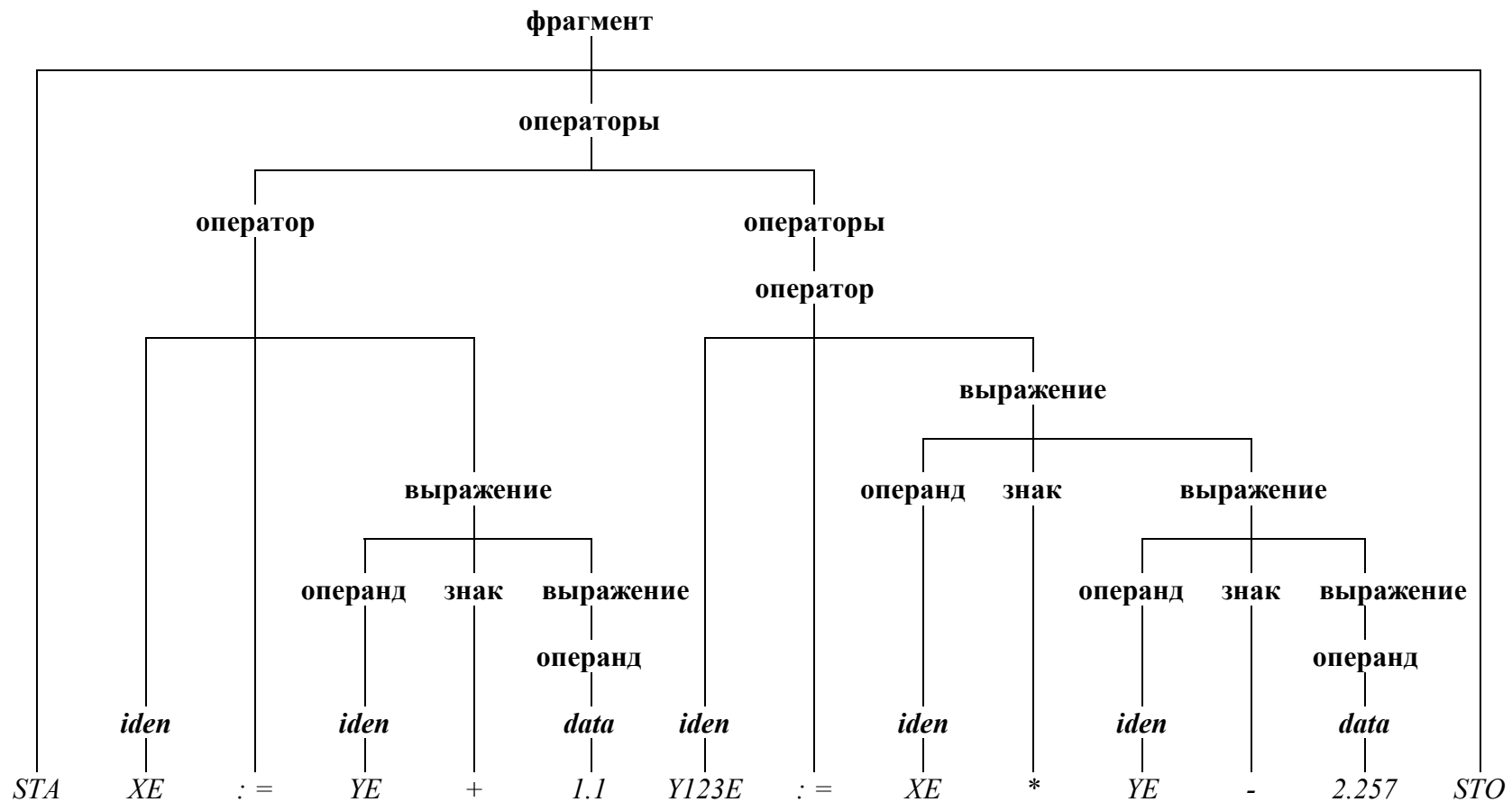


Рисунок 2.1 – Синтаксическое дерево фрагмента учебной программы

своего рода панацеей, следует попытаться переписать правила грамматики, чтобы привести ее к классу  $LL(k)$ . Основное внимание следует уделить устранению явной левой рекурсии.

Перепишем правило (1) в виде

$$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{тело программы} \rangle \quad (15)$$

Нетерминал  $\langle \text{тело программы} \rangle$  определим как

$$\langle \text{тело программы} \rangle ::= \langle \text{оператор} \rangle \langle \text{тело программы} \rangle \quad (16)$$

$$\langle \text{тело программы} \rangle ::= \text{STO} \quad (17)$$

Заметим, что грамматика после такой модификации, связанной с заменой правил (1) – (3) продукциями (15) – (17), описывает такие языковые конструкции, которые могут **не содержать** исполнимых операторов внутри операторных скобок STA, STO.

Следует признать, что реальные языки программирования «располагают» такими конструкциями как пустой оператор, пустой модуль (модуль-заглушка), что активно используется разработчиками программ при их отладке.

Правила (5) и (6) могут быть аналогично переписаны в виде

$$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \langle \text{продолжение} \rangle \quad (18)$$

$$\langle \text{продолжение} \rangle ::= \langle \text{знак} \rangle \langle \text{продолжение} \rangle \quad (19)$$

$$\langle \text{продолжение} \rangle ::= \langle \text{операнд} \rangle \quad (20)$$

$$\langle \text{продолжение} \rangle ::= \varepsilon \quad (21)$$

В выражении (21)  $\varepsilon$  обозначен пустой символ (цепочка),  $\varepsilon$ - грамматика является для алгоритмов разбора (кроме МП) неприемлемой. Предложенное решение имело бы смысл при **наличии явных разделителей операторов** (типа точки с запятой в Си или Pascal'e). Поэтому замена (5) и (6) на (18)-(21) нецелесообразна.

Окончательно  $LL(1)$ -грамматика примет вид.

$$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{тело программы} \rangle \quad (22)$$

$$\langle \text{тело программы} \rangle ::= \langle \text{оператор} \rangle \langle \text{тело программы} \rangle \quad (23)$$

$$\langle \text{тело программы} \rangle ::= \text{STO} \quad (24)$$

$$\langle \text{оператор} \rangle ::= \langle \text{idен} \rangle ::= \langle \text{операнд} \rangle \{ \langle \text{знак} \rangle \langle \text{операнд} \rangle \}_0^\infty \quad (25)$$

$$\langle \text{операнд} \rangle ::= \langle \text{idен} \rangle \quad (26)$$

$$\langle \text{операнд} \rangle ::= \langle \text{data} \rangle \quad (27)$$

$$\langle \text{знак} \rangle ::= + \quad (28)$$

$$\langle \text{знак} \rangle ::= - \quad (29)$$

$$\langle \text{знак} \rangle ::= * \quad (30)$$

$$\langle \text{знак} \rangle ::= / \quad (31)$$

Таким образом, в преобразованной грамматике десять правил (22)-(31) наряду со следующими элементами:

нетерминалы  $V_N = \{ \text{фрагмент}, \text{тело\_программы}, \text{оператор}, \text{операнд}, \text{знак} \}$ ;

терминалы  $V_T = \{ \text{STA}, \text{STO}, +, -, *, /, :=, \langle \text{idен} \rangle, \langle \text{data} \rangle \}$ ;

аксиома грамматики ФРАГМЕНТ.

Факт принадлежности грамматики к классу  $LL(1)$  и корректности построенной грамматики читателю предлагается проверить самостоятельно.

## 2.2 Построение нисходящего распознавателя по методу рекурсивного спуска с использованием синтаксических функций

При написании программы распознавателя предполагается, что каждому символу словаря грамматики соответствует функция, его опознающая. Указанные функции вызываются в последовательности, определенной правилами грамматики, результат их обработки – диагностическое сообщение об ошибке, и булева переменная, единичное значение которой соответствует удачному опознанию, а нулевое – ошибке. Прототип такой функции на языке С выглядит примерно так:

```
int S_symbol(void);
```

Для унификации можно имена функций начинать с имени S, а через подчеркивания указывать имя опознаваемого символа.

Одним из показателей удобства использования компилятора является наличие диагностических сообщений, позволяющих локализовать и осмыслить ошибку. В данном случае, сообщение целесообразно увязать с фактом неверного опознания того или иного символа.

В программе, соответствующей (22) – (31), будет присутствовать 14 синтаксических функций, в соответствии с объемом словаря грамматики, плюс одна – для выполнения факторизации. Тексты участков программы и отдельных синтаксических функций приводятся ниже.

```
int S_fragment(void), S_sta(void), S_telo(void),
S_operator(void), S_sto(void), S_iden(void),
S_operand(void), S_znak(void), S_data(void),
S_plus(void), S_minus(void), S_astra(void),
S_slash(void), S_prisv(void);
```

```
int znak_operand();
```

```
int code; /*текущий дескриптор лексем*/
```

```
void main(void)
```

```
{
```

```
FILE *Prog_code;
```

```
if ((Prog_code=fopen("...", "r")) == NULL)
```

```
    printf("\n Ошибка открытия");
```

```
else { fscanf(Prog_code, "%d", &code);
```

```
if (S_fragment() == 0)
```

```
    printf ("\n Синтаксическая ошибка");
```

```
else printf ("\n OK");
```

```
}
```

```
int S_fragment (void)
```

```

{
if (code == 100)
    return S_sta() * S_telo();
else {printf ("\n неверное начало программы");
      return 0;}
}

int S_telo (void)
{
    switch (code) {
    case 300:    return S_operator(); break;
    case 200:    return S_sto(); break;
    default:
                printf ("\n ожидалось начало оператора
                        или конец модуля");
                return 0;}
}

int S_operator (void)
{
if (code == 300)    return
s_iden() * s_prisv() * znak_operand()
    else {printf ("\n ожидался оператор");
          return 0;}
}

int znak_operand ( void)
{int response = 1;
  do while ( (code > 500) && (code < 505))
      response *= S_znak() * S_operand();
return response;
}

int s_operand (void)
{
    switch (code) {
    case 300:    return S_iden(); break;
    case 400:    return P_data( ); break;
    default:
                printf ("\n ожидалась константа или
                        переменная");
                return 0;
}
}

...
int S_prisv (void)
{

```

```

if (code == 600) {fscanf(Prog_code, "%d", &code);
                return 1;}
else { printf ("\n Нет знака присваивания");
      return 0;}
}

```

Примеры правильной и ошибочных тестовых последовательностей приводятся в таблице 2.2.

Таблица 2.2 – Тестовые последовательности

№	Анализируемая последовательность кодов	Результат
1	100 300 600 300 501 400 300 600 300 503 300 502 400 200	Ок
2	100 600 300 ...	Ожидался оператор. Синтаксическая ошибка
3	100 300 300 ...	Нет знака присваивания. Синтаксическая ошибка
4	100 300 600 501 ...	Ожидалась константа либо переменная. Синтаксическая ошибка

Логика работы нисходящего распознавателя данного типа такова, что обнаруживается только первая ошибка, до которой удалось добраться в процессе рекурсивного спуска. Обратите внимание, что синтаксический анализатор имеет дело с программой в виде кодов лексем, система кодирования которых была принята нами при построении сканера.

### 3. Построение восходящего распознавателя

Чтобы воспользоваться алгоритмом разбора, основанном на свойствах простого предшествования, необходимо решить следующие принципиальные вещи: описать язык программирования соответствующей грамматикой предшествования, разработать эффективную форму ее представления в памяти.

Первое – решается путем проверки существующей грамматики на отношение предшествования. Первоначальное построение грамматики ничем не отличается от процедуры, детально рассмотренной нами выше. Проверка сопряжена с построением отношений и лишь на последнем шаге проверяется их единственность для каждой пары символов [7, 8]. Так как данная процедура весьма громоздка, то желательно еще на первом этапе провести стратификацию продуктов грамматики. Показаниями для этой операции служит наличие нетерминального символа внутри цепочки правой части правила, который затем описывается с использованием левой или правой рекурсии [7].

### 3.1. Исследование грамматики на предшествование

В качестве исходного материала для исследования возьмем грамматику с продукциями (1) – (12), ранее нами составленную. Нетерминальный символ  $\langle \text{операторы} \rangle$ , стоящий посреди правила (1), праворекурсивно описывается правилом (3). Таким образом, между парой символов грамматики  $\langle \text{операторы} \rangle \dots \text{STO}$  будут иметь место сразу два отношения  $\langle \text{операторы} \rangle \bullet \text{STO}$  и  $\langle \text{операторы} \rangle = \text{STO}$ , что ясно видно на синтаксическом дереве (рисунок 2.1). В данном случае можно прибегнуть к стратификации (разделению) символов, то есть ввести дополнительный нетерминальный символ вместо конфликтного и описать его. При этом правило (1) примет вид

$$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{последовательность} \rangle \text{STO} \quad (32)$$

$$\langle \text{последовательность} \rangle ::= \langle \text{операторы} \rangle \quad (33)$$

Таким образом, мы приобрели несколько “бессмысленное”, с точки зрения порождаемого языка, правило, а словарь грамматики увеличился на один символ. Это ведет к увеличению вспомогательных и результирующей матриц, что, само по себе, нежелательно из-за роста объема вычислений.

Можно попытаться использовать грамматику (1) – (12) с заменой правил (1) – (3) правилами (15) – (17). При этом объем словаря не изменится, грамматика станет “осмысленной”, явного проявления неоднозначности отношений предшествования не отмечается. Окончательно получим:

$$\langle \text{фрагмент} \rangle ::= \text{STA} \langle \text{тело программы} \rangle$$

$$\langle \text{тело программы} \rangle ::= \langle \text{оператор} \rangle \langle \text{тело программы} \rangle \mid \text{STO}$$

$$\langle \text{оператор} \rangle ::= \langle \text{iden} \rangle := \langle \text{выражение} \rangle$$

$$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \mid \langle \text{операнд} \rangle \langle \text{знак} \rangle \langle \text{выражение} \rangle$$

$$\langle \text{операнд} \rangle ::= \langle \text{iden} \rangle \mid \langle \text{data} \rangle$$

$$\langle \text{знак} \rangle ::= + \mid - \mid * \mid /$$

Имеем: нетерминальные символы  $V_n = \{\text{фрагмент}, \text{тело программы}, \text{оператор}, \text{выражение}, \text{операнд}, \text{знак}\}$  – 6 символов; терминальные символы и условно терминальные  $V_t = \{\text{STA}, \text{STO}, \text{iden}, \text{data}, :=, +, -, *, /\}$  – 9 символов. Таким образом, мощность словаря грамматики равна 15.

Составим матрицы отношений EQ (соседние символы), F (начинается на), L (заканчивается на) по правилам [7].

1. EQ (equivalence):  $\text{EQ}(R, S) = 1$ , при  $E \rightarrow uRSv$ , где  $E \in V_n$ ,  $R, S \in V$ ,  $u, v \in V^*$ .

2. F (first):  $F(\langle R \rangle, S) = 1$ , при  $R \rightarrow Sv$ , где  $R \in V_n$ ,  $S \in V$ ,  $v \in V^*$ .

3. L (last):  $L(\langle R \rangle, S) = 1$ , при  $R \rightarrow Su$ , где  $R \in V_n$ ,  $S \in V$ ,  $u \in V^*$ .

При составлении матрицы рекомендуется: а) группировать нетерминальные символы к началу таблицы; б) использовать один и тот же порядок группировки для заголовков столбцов и строк; в) с целью экономии времени на разметку заголовков, совместите все три матрицы в одной, как это показано в таблице 2.3. Буквами e, f, l обозначены соответствующие значения.



Транзитивное замыкание отношения заданного булевой матрицей  $A$  вычисляется по правилу

$A^+ = A^1 \cup A^2 \cup A^3 \cup \dots$ , где  $A^1 = A$ ,  $A^2 = A * A$  и т.д.

Причем возведение продолжается до тех пор, пока результат очередного возведения в степень матрицы  $A$  не станет равным нулю либо результату, полученному на какой-либо предыдущей итерации.

Матрицы большой размерности могут быть получены с использованием алгоритма S.Warshall'a [7], реализация которого на Pascal'е приводится ниже.

```

Procedure Warchall;
Var i, j, k: integer;
Begin
  for i:=1 to n do
    for j:=1 to n do
      Aplus[i, j] := A[i, j];
  for i:=1 to n do Aplus[i, i] := 0;
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        if (not Aplus[i, j]) then
          Aplus[i, j] := Aplus[i, k] and Aplus[k, j];
End;
```

Окончательно отношения  $R < \bullet S$  вычисляют по выражению  $EQ \times F^+$ , а отношение  $R \bullet > S (L^+)^T \times EQ \times (I + F^+)$ , где  $I$  – единичная матрица.

После выполнения расчетов, все три отношения сводят в одну матрицу, при этом выясняется единственность отношений. Указанные вычисления читателю предлагается проделать самостоятельно. Результат вычисления отношений представлен в таблице 2.4.

Для программной реализации символы отношений могут быть закодированы. Например,  $1 - < \bullet$ ,  $2 - \equiv$ ,  $3 - \bullet >$ . А для тех позиций матрицы, где отношения между символами не определено, поставить коды ошибок.

Из таблицы отношений видно, что разработанная грамматика является грамматикой простого предшествования.

С целью минимизации размеров матрицы предшествования прибегают к построению функций предшествования [7, 8]. При этом размер матрицы уменьшается с  $n \times n$  на  $2 \times n$  – по числу функций, каждая из которых есть одномерный массив.

Процедура построения функций показана на примере [9]. К сожалению, факт наличия или отсутствия для грамматики функций предшествования обнаруживается только после громоздких матричных вычислений. По существующему мнению, например [7], замена матрицы отношения функциями предшествования препятствует раннему обнаружению, точной диагностике и нейтрализации возникающей ошибки.



Таблица 2.4 – Отношение предшествования для разработанной грамматики

R \ S	фрагмент	Тело программы	Оператор	выражение	операнд	знак	STA	STO	iden	data	:=	+	-	*	/
фрагмент															
Тело программы															
Оператор		≡	< •					< •	< •						
выражение		• >	• >					• >	• >						
операнд		• >	• >			≡		• >	• >			< •	< •	< •	< •
знак				≡	< •				< •	< •					
STA		≡	< •					< •	< •						
STO															
iden						• >					≡	• >	• >	• >	• >
data						• >						• >	• >	• >	• >
:=				≡	< •				< •	< •					
+				• >	• >				• >	• >					
-				• >	• >				• >	• >					
*				• >	• >				• >	• >					
/				• >	• >				• >	• >					

## 1.2. Построение восходящего распознавателя

Для реализации восходящего распознавателя, управляемого отношением простого предшествования, существуют алгоритмы как магазинного распознавателя [3, 4], так и логического его варианта [5, 7].

Учитывая квалификацию читателя, выполнить реализацию этих алгоритмов не составит труда. Независимо от выбранного алгоритма, необходимо выполнять операцию сопоставления некоторой строки, находящейся в магазине и заканчивающейся вершиной, которая претендует на роль основы синтаксического разбора с правой частью правила. Для этих целей эффективно представить грамматику в виде одномерного массива и использовать алгоритм КМП (Кнута, Мориса и Пратта) [6] поиска подстроки в строке.

Например, кодированное представление нашей грамматики имеет вид, помещенный ниже

701	800	100	702	900	
702	800	200	900		
702	800	300	600	704	900
703	800	705	706	704	900
704	800	705	900		

705	800	300	900
705	800	400	900
706	800	501	900
706	800	502	900
706	800	503	900
706	800	504	900

Дополнительными кодами обозначены нетерминалы: 701 – фрагмент, 702 – тело программы, 703 – оператор, 704 – выражение, 705 – операнд, 706 – знак; и метасимволы: 800 – ::= и 900 – конец правила.

Ошибки, обнаруженные распознавателем, бывают двух видов. Первая обнаруживается, когда между двумя соседними символами анализируемой строки отсутствует отношение предшествования на этапе выполнения магазинных операций. В данном случае выдается диагностическое сообщение вида “Символы R и S не должны стоять рядом”.

В данном сообщении символ R не обязательно терминальный. Он может представлять собой результат подстановки, выполненной на предыдущем шаге.

Таким образом, строки диагностической матрицы отношений 1,2,8 и столбец 1 могут быть заполнены кодом ошибки 4 – “Неверная структура программы”, и столбец 7 в строке 3, а также строки с 9 по 15 в столбцах 7 и 8. Оставшиеся позиции в строке 7 заполняем кодом 5 – “Ошибка компоновки тела модуля”, незаполненные позиции выше 7-ой строки отметим кодом 6 – “Ошибка составления выражения”, этим же символом заполним пустые позиции строк 9 и 10. Оставшуюся часть таблицы заполним кодом ошибки 7 – “Недопустимая комбинация знаков”. Результаты показаны в таблице 2.5.

Таблица 2.5 – Диагностическая матрица распознавателя

4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
4	2	1	6	6	6	4	1	1	6	6	6	6	6	6
4	3	3	6	6	6	6	3	3	6	6	6	6	6	6
4	3	3	6	6	2	6	3	3	6	6	1	1	1	1
4	6	6	2	1	6	6	6	1	1	6	6	6	6	6
4	2	1	5	5	5	5	1	1	5	5	5	5	5	5
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
4	6	6	6	6	1	4	4	6	6	2	3	3	3	3
4	6	6	6	6	1	4	4	6	6	6	3	3	3	3
4	7	7	2	1	7	4	4	1	1	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7

Текст программы анализатора, как не играющий в нашем повествовании смысловой роли, а относящийся к области искусства программирования, мы представим конспективно.

```

...     ...     ...
# define VOC_POWER 16

...     ...     ...

// Определение номера строки или столбца диагностической
// таблицы
int NumberColRow(int *cod)
{
    switch (cod) {
        case 100: return 6; break;    //STA
        case 200: return 7; break;    //STO
...     ...     ...
        case 600: return 10; break;   // :=
        case 701: return 0; break;    // <фрагмент>
...     ...     ...
        case 706: return 5; break;    // <знак>
        case 999: return 15; break;   // #
        default;;
    }
}

// Поиск правила, правая часть которого совпадает
// с основой
int FindndRule(int *Grammar, int *stack, int
stack_pointer, int head_pointer)
{
...     ...     ...
}

...     ...     ...

// Поиск местоположения головы основы в стеке
int FindPointer(int *stack, int stack_pointer)
{
...     ...     ...
}

void main()
```

```

{
// Диагностическая таблица
int SR[VOC_POWER][VOC_POWER]={
//////// 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/*0*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 99},
/*1*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1},
/*2*/ {4, 2, 1, 6, 6, 6, 4, 1, 1, 6, 6, 6, 6, 6, 6, 1},
/*3*/ {4, 3, 3, 6, 6, 6, 6, 3, 3, 6, 6, 6, 6, 6, 6, 1},
/*4*/ {4, 3, 3, 6, 6, 2, 6, 3, 3, 6, 6, 1, 1, 1, 1, 1},
/*5*/ {4, 6, 6, 2, 1, 6, 6, 6, 1, 1, 6, 6, 6, 6, 6, 1},
/*6*/ {4, 2, 1, 5, 5, 5, 5, 1, 1, 5, 5, 5, 5, 5, 5, 1},
/*7*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1},
/*8*/ {4, 6, 6, 6, 6, 1, 4, 4, 6, 6, 2, 3, 3, 3, 3, 1},
/*9*/ {4, 6, 6, 6, 6, 1, 4, 4, 6, 6, 6, 3, 3, 3, 3, 1},
/*11*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*10*/ {4, 7, 7, 2, 1, 7, 4, 4, 1, 1, 7, 7, 7, 7, 7, 1},
/*12*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*13*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*14*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*15*/ {3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 99}
};

```

```

// Продукции формальной грамматики
int *G[NT_POWER]={
        701, 800, 100, 702, 900,
        702, 800, 200, 900,
        702, 800, 300, 600, 704, 900,
        703, 800, 705, 706, 704, 900,
        704, 800, 705, 900,
        705, 800, 300, 900,
        705, 800, 400, 900,
        706, 800, 501, 900,
        706, 800, 502, 900,
        706, 800, 503, 900,
        706, 800, 504, 900};

```

```

/// Начальные установки
magazine[0] = 999; // Код знака "#", выталкиватель стека
mp = 0;           // Указатель заполнения магазина (стека)
. . .
// Такт работы распознавателя (управляющее устройство)
NoStrk = NumberColRow(magazin[mp]); // Адресация таблицы
NoStlb = NumberColRow(code);

```

```

switch (SR[NoStrk][NoStlb]){
  case 1: // Операция переноса для случая <•
    mp++;
    magazine[mp] = 1; // Ограничитель начала фразы
    mp++;
    magazine[mp] = code;
    nf_pointer = mp; // Запоминаем голову последней
                     // простой фразы
    break;
  case 2: // Операция переноса для случая эквивалентности
    mp++;
    magazine[mp]=code;
    break;
  case 3: // Операция свёртки, основа в магазине
    NonTerm = FindRule(GR, magazine, mp, nf_pointer);
    if (SR[NumberColRow(magazin[nf_pointer - 1])]
        [NumberColRow(NonTerm)] == 1) // <•
    {
      magazin[nf_pointer] = NonTerm;
      mp = nf_pointer;
    }
    else // отношение эквивалентности
    {
      nf_pointer --;
      magazin[nf_pointer] = NonTerm;
      mp = nf_pointer;
      nf_pointer = FindPointer(magazine, mp);
    }
    break;
  case 4:
    printf ("Invalid program structure.");
    getch();
    return 0;
    break;
  case 5:
    printf ("Module composition mistake.");
    getch();
    return 0;
    break;
  case 6:
    printf ("Invalid statement.");
    getch();
    return 0;
    break;
}

```

```

case 7:
    printf ("Illegal sign combination.");
    getch();
    return 0;
    break;
case 99:
    printf ("Syntax checked. Ok.");
    getch();
    return 0;
    break;
default:
    printf ("Unknown Error in syntax.");
    getch();
    return 0;
}

}

```

Результат выполнения отдельных текстовых последовательностей представлен в таблице 2.6.

Таблица 2.6 – Результаты выполнения контрольных примеров

№	Анализируемый фрагмент	Диагностическое сообщение
1	100 600 300...	“Служебное слово STA” и “знак присваивания := ” несовместны ERR5: ошибка компоновки тела модуля (Module composition mistake)
2	100 300 300...	“имя переменной” и ”имя переменной” не совместны ERR6: ошибка составления выражения (Invalid statement)
4	100 300 600 501...	“знак присваивания” и ”знак +” не совместны ERR7: недопустимая комбинация знаков (Illegal sign combination)

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Алгоритмический язык АЛГОЛ 60. Модифицированное сообщение. – М.: Мир, 1982. – 72 с.
2. Алкок Д. Язык Паскаль в иллюстрациях. / Д. Алкок. – М.: Мир, 1991. – 72 с.
3. Ахо А. Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ./ А. Ахо, Дж. Ульман. – М.: Мир, 1978. – 619 с.
4. Ахо А. Компиляторы: принципы, технологии и инструменты/ А. Ахо, Р. Сети, Дж. Ульман. . – М.: Издательский дом «Вильямс», 2002. – 412 с.
5. Вайнгартен Ф. Трансляция языков программирования / Ф. Вайнгартен - М.: Мир, 1977.-190 с.
6. Вирт Н. Алгоритмы и структуры данных / Н. Вирт. - М.: Мир, 1989. - 360 с.
7. Грис Д. Конструирование компиляторов для ЦВМ / Д. Грис - М.: Мир, 1975.-544 с.
8. Льюис Ф. Теоретические основы проектирования компиляторов. / Ф. Льюис, Д. Розенкранц, Р. Стирнз. –М.: Мир, 1979. – 564 с.
9. Методические указания к выполнению лабораторных и контрольных работ по дисциплине “Теоретические основы построения компиляторов” для студентов всех форм обучения основного профиля 09.03.02 – “Информационные системы и технологии”. Основные алгоритмы [Текст] / Разраб. В.Ю. Карлусов. – Севастополь: Изд-во СевГУ, 2015. – 44 с.
10. Рейуорд-Смит В.Дж. Теория формальных языков. Вводный курс / В.Дж. Рейуорд-Смит- М.: Мир, 1986.-128 с.
11. Хантер Р. Основные концепции компиляторов. / Р.Хантер. – М.: Издательский дом «Вильямс», 2002. – 256 с.
12. Хопкрофт Дж. Введение в теорию автоматов, языков и вычислений. / Дж. Хопкрофт, Р. Мотвани, Дж. Ульман. – М.: Издательский дом «Вильямс», 2002. – 528 с.

## ПРИЛОЖЕНИЕ А

### (обязательное)

Варианты, по усмотрению преподавателя, могут быть выданы по номерам зачётной книжки с использованием таблицы А.1. Содержимое первой цифры варианта (описание синтаксиса идентификатора учебной программы) раскрывается в таблице А.2, второй (описание синтаксиса констант) – в А.3, третья (служебные слова, разделители и фрагменты программ) – в А.4.

Таблица А.1 – Список вариантов сквозного контрольного задания к контрольным работам № 1 и 2 по теории синтаксического анализа

1-я цифра	2-я цифра									
	1	2	3	4	5	6	7	8	9	0
1	1.1.15	14.8.2	3.8.8	13.7.4	15.6.5	16.4.3	17.3.6	1.2.7	2.1.8	3.8.9
2	12.2.10	2.7.11	13.3.12	4.2.13	5.7.14	6.6.1	9.5.1	10.4.9	13.3.2	4.2.4
3	8.3.5	11.6.6	3.4.14	9.1.7	12.6.8	5.7.9	11.4.9	12.3.10	14.2.11	5.1.12
4	7.4.14	10.5.13	6.5.15	4.8.1	14.8.2	11.7.4	15.6.5	16.5.6	17.4.3	6.3.7
5	17.8.8	1.7.9	2.6.10	3.7.1	5.2.2	15.1.4	8.4.14	1.2.5	2.3.6	7.4.10
6	16.7.7	11.4.14	15.5.8	14.6.9	13.5.10	6.7.11	16.6.12	10.8.13	3.1.14	8.2.1
7	7.6.2	8.3.4	9.6.5	10.5.6	11.3.7	12.4.10	7.5.8	17.6.9	11.7.10	4.8.11
8	6.5.12	5.2.13	4.7.14	3.4.1	2.1.2	1.2.4	17.3.5	8.4.3	1.5.6	12.6.7
9	9.4.10	10.1.8	11.8.9	12.3.10	13.7.11	14.8.12	15.1.13	16.2.14	9.3.1	2.4.9
0	8.3.2	7.2.4	7.1.5	6.2.6	5.5.7	4.6.8	3.7.9	2.8.10	1.6.11	10.9.12



Таблица А.2 – - Типы идентификаторов

№ вариан-та	Описание типа
(1)	(2)
1	Содержит чередующиеся пары букв и цифр, заканчивается последовательностью символов "1"
2	Содержит чередующиеся буквы и цифры, заканчивается последовательностью символов "0"
3	Если встречается символ "a", то за ним всегда следует символ "c"
4	За встреченным символом "b" следует последовательность нулей и единиц
5	Оканчивается символом "m", которому не предшествует буква
6	Оканчивается символом "s", которому не предшествует цифра
7	Включает последовательность "abs"
8	Содержит последовательность символов "1" и оканчивается цифрой
9	Содержит последовательность символов "0" и оканчивается буквой
10	Оканчивается последовательностью цифр и включает символ "a"
11	Оканчивается последовательностью букв и включает последовательность "11"
12	Оканчивается символом "0" и включает последовательность "0100"
13	Следом за начальной буквой идентификатора следует "111"
14	Цифры, встречающиеся в идентификаторе, если они присутствуют в его записи, упорядочены по возрастанию
15	Цифры, включенные в состав идентификатора, сгруппированы в одну последовательность
16	В идентификаторе не встречается более трех букв, идущих подряд
17	В идентификаторе не встречается более трёх цифр, идущих подряд
18	Никакие две одинаковые буквы или цифры не стоят рядом

Таблица А.3 – Типы констант

№ варианта	Тип	Пояснения	Формат
1	F	С фиксированной точкой	$\pm 000.000$
2	E	С плавающей точкой	$\pm 000.000E\pm 00$
3	B	Двоичная	$\pm 10...001B$
4	C	Символьная	'a...Z0...9'
5	X	Шестнадцатеричная	$\pm 0x12...9a...f$
6	O	Восьмеричная	$\pm 0o12...7$
7	D	Удвоенная, с плавающей точкой	$\pm 000.000D\pm 000$
8	L	Удвоенная, целая	$\pm 1001...L$

Примечание Для типа F задаются: максимальный размер целой части P и максимальный размер дробной части q. Для типов B,C,X,O,L - задается число цифр в записи. Тип E имеет максимально 6 цифр в мантиссе и две в показателе степени. Тип D имеет максимально 16 цифр в мантиссе и 3 в показателе степени.

Таблица А.4 – Фрагмент программ для анализа

№ варианта	Служебные слова	Разделители		Фрагмент программы для анализа
		одно-литерные	двулитерные	
(1)	(2)	(3)	(4)	(5)
1	BEGIN END	; = * /		BEGIN; x=4; B=4.5*x+3; END;
2	START STOP	* + , -	:=	START, x:=-0.5, y:=3*x+1, STOP
3	PUSK OFF	& ! =	\\	PUSK \\ x='ABC' \\ y=x&'A'!B \\ OFF
4	PRINT SCAN	, ; + - ( )		PRINT (A+B,'C'); SCAN (F)

продолжение таблицы А.4

(1)	(2)	(3)	(4)	(5)
5	IF THEN	( ) & +	== :=	IF (A==B) & (C==0x0f) THEN K:=K+1;
6	DO WHILE END	= - + , ( )	<= >=	WHILE (K<=0o171) DO K=K+1, A=A-1, END,
7	FOR TO	( ) = ; * /		FOR (I=1) TO 4 ( K=K*5; C=C/2E+01; );
8	CASE SWITCH	{ } ( ) + - ; = :		K=5; C=5+x; SWITCH(C) { case 1: K=K+1; case 2: K=K+2; }
9	CHAR INTEGER	, ; = ( )	!! << >>	CHAR (A,B); INTEGER (C); A='SANTAS DUMON'; B=A!!A;
10	PROC END	= ,	&& !! << >>	PROC, A=0xA1011, B=0xae13, C=A&&B, END,
11	READ WRITE	( ) * / - + , \$		WRITE (A*B, C+23) \$ READ (D) \$

продолжение таблицы А.4

(1)	(2)	(3)	(4)	(5)
12	IF THEN ELSE	< > = + - ( ) & :		IF (A>B & C<0) THEN (A=A+B+7) ELSE (A=A-B; C=D+B);
13	DO UNTIL	( ) = + - * / < > #	<>	DO x=3+y # z=2/x+1 UNTIL(z<>0.01)#
14	FUNC FINAL	: \ ! { } ( )	:= && !!	A: FUNC (x)\ X:=(x&&0xfe)\ A:=!(x!!0x13)\ FINAL:
15	WAIT SIGNAL	( ) > = + %	>= <=	WAIT(S) K=K+5.3 % C=K+C+1 % SIGNAL(S<=0xf)

Примечание – Вид разбираемых инструкций, приводимых в графе (5) таблицы А.4, весьма условен в части идентификаторов и констант.