

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ КОМПИЛЯТОРОВ

Учебно-методическое пособие

к выполнению вычислительного практикума и контрольных работ
по дисциплине для студентов основного профиля
09.03.02 – «Информационные системы и технологии» всех форм
обучения



Севастополь
2022

УДК 004.4'422:519.713(076.5)

ББК 32.815я73

T338

Рецензенты:

В. С. Чернега, кандидат техн. наук, доцент кафедры Информационных систем:

М. В. Заморёнов, кандидат техн. наук, доцент кафедры Информационных технологий и компьютерных систем.

Составители: В. Ю. Карлусов, С. А. Кузнецов

T338 Теоретические основы построения компиляторов: учебно-методическое пособие к выполнению вычислительного практикума и контрольных работ по дисциплине для студентов основного профиля 09.03.02 – «Информационные системы и технологии» всех форм обучения / Севастопольский государственный университет; сост.: В. Ю. Карлусов, С. А. Кузнецов. – Севастополь : СевГУ, 2022. – 47 с.

Цель методического пособия: обеспечение студентов дидактическим материалом для качественного выполнения лабораторных и контрольных работ; по разделам: формальные языки и грамматики, элементы теории конечных автоматов, отношения предшествования в приложении к задачам анализа программ на алгоритмических языках.

УДК 004.4'422:519.713(076.5)

ББК 32.815я73

Методические указания рассмотрены и утверждены на заседании кафедры Информационных систем,
протокол № 06 от 05 февраля 2022 г.

Методическое пособие рассмотрено и рекомендовано к изданию на заседании Учёного Совета Института информационных технологий и управления в технических системах от 18 апреля 2022 г. года, протокол № 06.

Ответственный за выпуск: заведующий кафедрой Информационных систем, канд. физ.-мат. наук, доцент И. П. Шумейко

© СевГУ, 2022

© Карлусов В. Ю., Кузнецов С. А., 2022

Издательский номер №№ 149/22

Содержание

Общие положения.....	4
1. Построение минимального детерминированного конечного автомата (МДКА) по регулярному выражению	5
1.1. Краткие теоретические сведения	5
1.2. Пример построения МДКА	5
2. Построение лексического анализатора на базе конечного автомата	11
2.1.Краткие теоретические сведения	11
2.2. Пример построения МДКА для сканера	12
2.2.1. Определение лексем и назначение им кодов	12
2.2.2. Построение регулярного выражения по варианту исходных данных	13
2.2.3. Построение функции переходов МДКА	14
2.3 Методология разработки программы сканера	19
3. Построение синтаксических анализаторов	22
3.1. Краткие теоретические сведения	22
3.2. Построение нисходящего распознавателя	23
3.2.1. Построение $LL(k)$ -грамматики	23
3.2.2 Реализации нисходящего распознавателя методом рекурсивного спуска	27
3.3. Построение восходящего распознавателя	29
3.3.1. Исследование грамматики на предшествование	30
3.3.2. Программная реализация восходящего распознавателя	33
Заключение.....	39
Библиографический список.....	39
Приложение А. Варианты к построению МДКА	41
Приложение Б. Варианты к построению лексического и синтаксического анализаторов	42

ОБЩИЕ ПОЛОЖЕНИЯ

Настоящее методическое пособие представляет примеры выполнения лабораторных работ для очной формы обучения и контрольной работы для заочной формы обучения.

Код варианта является общим для выполнения работ и выбирается, используя две последние цифры зачётной книжки, по таблицам, содержащимся в приложениях А и Б для заочной формы обучения. Для дневной формы обучения код выдаётся преподавателем.

Приложение А позволяет сконструировать регулярное выражение, описывающее конечный автомат, который студент должен реализовать.

В приложении Б по шифру варианта выбирается сквозное задание, используемая при решении задач лексического анализа, восходящего и нисходящего синтаксического анализа

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

Первый код относится к описанию идентификаторов языка учебной программы, содержащемуся в таблице Б.2. Следующая цифра относится к типу констант, который описывается в таблице Б.3. Последнее число позволяет определить по таблице Б.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 отчётов по контрольным и лабораторным работам предъявляются следующие требования.

Для задания, посвящённого синтезу МДКА, в отчёт включаются минимизация по регулярному выражению, по таблице, граф конечного автомата.

Для задачи построения лексического анализатора 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 студенты ЗФО демонстрируют на сессии.

Для задач синтаксического анализа 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 студенты ЗФО демонстрируют на сессии.

1. ПОСТРОЕНИЕ МИНИМАЛЬНОГО ДЕТЕРМИНИРОВАННОГО КОНЕЧНОГО АВТОМАТА (МДКА) ПО РЕГУЛЯРНОМУ ВЫРАЖЕНИЮ

1.1. Краткие теоретические сведения

Регулярной грамматике соответствует конечный автомат (КА), множество входных цепочек которого соответствует множеству цепочек порождаемых грамматикой.

Конечным автоматом (КА) формально называют совокупность следующих объектов (компонентов):

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

где Q – конечное множество неструктурируемых состояний автомата; Σ – непустое множество (алфавит) входных символов (букв или литер); δ – функция переходов КА, определяемая на декартовом произведении $Q \otimes \Sigma$; $q_0 \in Q$ – начальное состояние КА; $F \subset Q$ – множество заключительных состояний КА.

Назначение функции перехода состоит в определении последующего состояния КА, в зависимости от текущего состояния, по лите на его входе.

Множество всех цепочек, допускаемых КА, называется **регулярным множеством**. Оно, в ряде случаев, может быть описано регулярным выражением.

Таким образом, ставится задача построения функции переходов КА δ по регулярному выражению. В программе функция переходов представляется в виде диагностической таблицы, строки которой ставятся в соответствие алфавиту входных символов (литер) Σ , а столбцы – множестве состояний КА Q .

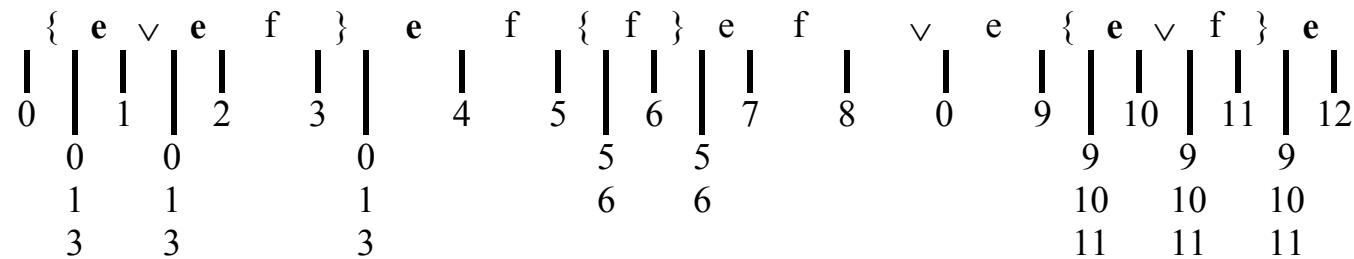
Правила написания регулярных выражений рассмотрены в [9, 10, 12], а практикумы построения и минимизация КА по регулярному выражению приведены в [9].

Указанные процедуры основаны на разметке регулярного выражения, появлению системы основных и предосновных мест. Разметка трансформируется в табличное представление функции переходов.

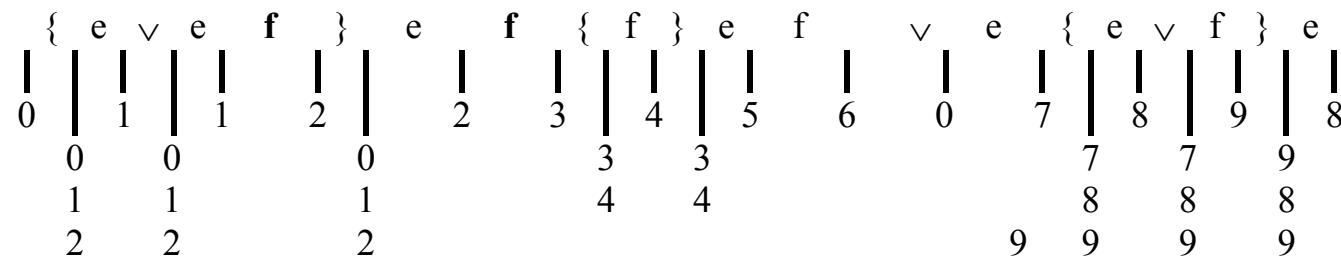
1.2. Пример построения МДКА

Выполнить построение КА, воспринимающего цепочки литер, описываемые выражением $\{e \vee f\}ef\{f\}ef \vee e\{e \vee f\}e$.

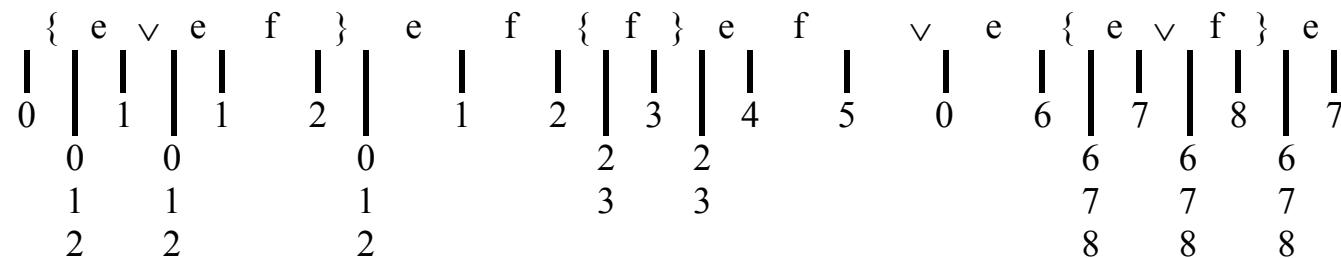
Разметка и ход упрощения показаны на рисунке 1.1, жирным шрифтом показаны позиции, подпадающие под правило минимизации. По окончании упрощения построим функцию переходов (таблица 1.1).



а) Первоначальная разметка регулярного выражения



б) Разметка после первого шага упрощения



в) Окончательный результат упрощения

Рисунок 1.1 – Ход упрощения регулярного выражения

Таблица 1.1. Представление функции переходов по результатам упрощения

Q	0	1	2	3	4	5	6	7	8
e	1, 6	1	1,4	4			7	7	7
f		2	3	3	5		8	8	8

Автомат, с функцией перехода, представленной в таблице 1, будет недетерминированным конечным автоматом (НДКА). Это видно по столбцам 0 и 2, в строке e , где образ функции перехода неоднозначен. Для построения детерминированного конечного автомата (ДКА) по НДКА, существуют соответствующие алгоритмы [3, 4, 10 – 12], примеры применения которых показаны в [7]. Для нашего примера, процесс приведение НДКА к ДКА проиллюстрируем в виде деревьев на рисунке 1.2.

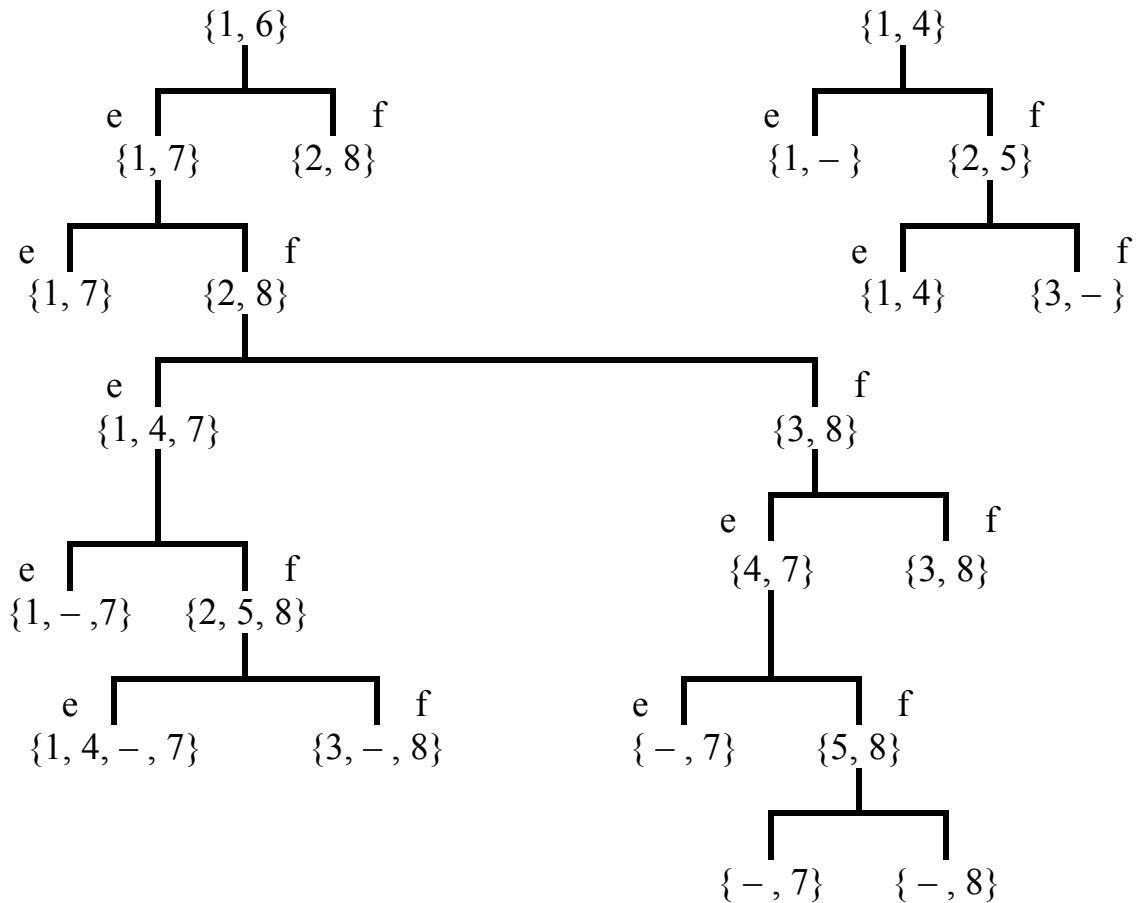


Рисунок 1.2 – Деревья приведения КА к однозначности функции переходов

Произвольно пронумеровав листья деревьев уникальными номерами, начиная с 9: $\{1, 4\} \rightarrow 9$, $\{2, 5\} \rightarrow 10$, $\{1, 6\} \rightarrow 11$, $\{1, 7\} \rightarrow 12$, $\{2, 8\} \rightarrow 13$, $\{1, 4, 7\} \rightarrow 14$, $\{3, 8\} \rightarrow 15$, $\{2, 5, 8\} \rightarrow 16$, $\{5, 8\} \rightarrow 17$, $\{4, 7\} \rightarrow 18$, дополним таблицу 1.1 соответствующими столбцами.

Таблица 1.2. Функция переходов ДКА с сигналами состояний

<i>S</i>	н	р	р	р	р	к	р	к	р	р	к	р	к	р	к	р	к	к	к
<i>Q</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
e	11	1	9	4			7	7	7	1	9	12	12	14	12	18	14	7	7
f		2	3	3	5		8	8	8	10	3	13	13	15	16	15	15	8	17

Отыщем состояния ДКА, недостижимые из начального (нулевого) состояния. Алгоритм поиска излагается в [4, 10], суть его проста и поддаётся умственному спрямлению: используя функцию переходов, строим множество достижимых состояний КА. Состояния, не вошедшие, во множество достижимых состояний, образуют множество недостижимых состояний.

Поиск достижимых состояний: $0 \rightarrow 11 \rightarrow 12, 13 \rightarrow 14, 15 \rightarrow 16, 18 \rightarrow 7, 17 \rightarrow 8$.

Таким образом, множество достижимых состояний КА $\{0, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18\}$, в таблице 2 их номера выделены жирным шрифтом. Состояния $\{1, 2, 3, 4, 5, 9, 10\}$ недостижимы из начального состояния ни для какой цепочки символов входного алфавита.

Выполним минимизацию по таблице переходов КА. Для этого введём значения сигналов: н – начальное состояние, р – рабочее состояние, к – конечное состояние конечного автомата.

Помимо явных конечных состояний 5 и 7 (см. разметку на рисунке 1, в), в ходе приведения НДКА к КА, образовались множества состояний, в качестве элементов которых присутствуют 5 и 7. Сигналы помещены в таблице 1.2 в верхней строке *S*.

Можно объединять состояния, соответствующие столбцам таблицы с одинаковыми сигналами и с одинаковым содержимым, таковыми являются состояния №№ 7 и 17 (они в таблице выделены серым цветом).

Для корректного преобразования таблицы функции переходов, необходимо выполнить перенумерацию столбцов и их содержимого с учётом объединения и исключения.

Состояния $\{0\} \rightarrow 0, \{7, 17\} \rightarrow 1, 8 \rightarrow 2, 11 \rightarrow 3, 12 \rightarrow 4, 13 \rightarrow 5, 14 \rightarrow 6, 15 \rightarrow 7, 16 \rightarrow 8, 18 \rightarrow 19$.

Таблица 1.3 – Функция переходов МДКА

<i>S</i>	н	к	р	р	к	р	к	р	к	к
<i>Q</i>	0	1	2	3	4	5	6	7	8	9
e	3	1	1	4	4	6	4	9	6	1
f		2	2	5	5	7	8	7	7	1

Функции переходов, представленной в таблице 1.3, соответствует ориентированный граф конечного автомата, представленный ниже. Начальное со-

стояние отмечено стрелкой, заключительные – представляются прямоугольниками, остальные – круглые.

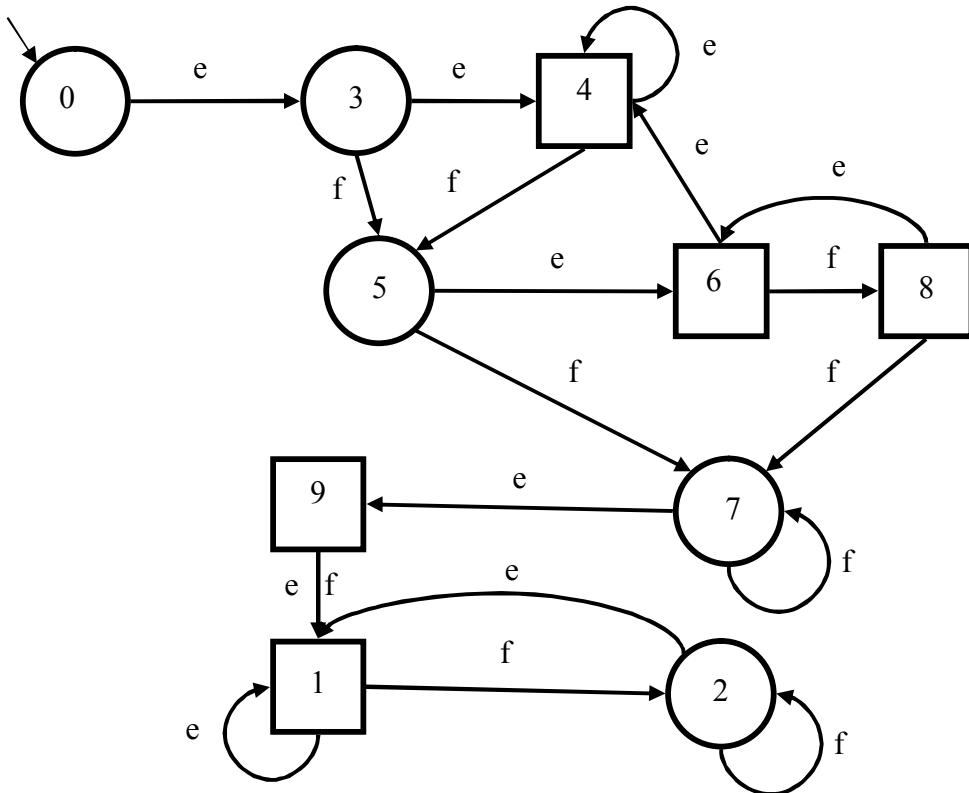


Рисунок 1.3 – Граф МДКА

Для программной реализации МДКА нужно доопределить функцию переходов: элементу (столбец 0, строка f) поставим код ошибки 99, сопоставляемый с диагностическим сообщением “Неправильная голова цепочки”. Текст программы приводится на рисунке 1.4.

Следует отметить, что регулярное множество, приведённое в примере, допускает два класса цепочек:

- один класс цепочек предписывает хвост ef для правильной цепочки и присутствия хотя бы одной пары ef внутри;
- второй определяет хвост e и, в принципе, имеет бесконечное подмножество цепочек, отличающихся длиной, состоящих только из этой литеры, и не включающих f.

Если не стоит задача классификации цепочек, а только проверка принадлежности, то синтезированный нами автомат её решает. В противном случае, необходимо ввести новые сигналы выходов на этапе минимизации по таблице, и провести эту процедуру заново.

В этом случае число состояний МДКА, по сравнению с построенным нами, возросло, что находится в полном соответствии с теоремой Майхилла-Нерода [10].

```

main ()
{
char c; /* текущая литера */
int i, n;
file * text;
int mp[2][10] = // управляющая таблица
{
{3, 1, 1, 4, 4, 6, 4, 9, 6, 1},
{99, 2, 2, 5, 5, 7, 8, 7, 7, 1}
};
text = fopen ('xx.txt', 'r'); // цепочки через пробел
i=0; // i - номер состояния КА
n=0; // n - число правильных цепочек в файле
while ((c = getc(text)) != EOF)
{
switch (c)
{
case 'e': i = mp[0][i]; break;
case 'f': i = mp[1][i]; break;
case ' ': {
    if (i ==1) | (i ==4) |
        (i ==6) | (i ==8) |
        (i ==9) {i = 0; n++;}
    break;
}
default: {i = 0;
printf("Не символ алфавита");break;
}
}
if (i ==99) { i = 0;
printf("Неправильная голова цепочки");}
}
printf("Количество цепочек =%d", n);
fclose (text);
}

```

Рисунок 1.4 – Один из возможных вариантов программной реализации МДКА

2. ПОСТРОЕНИЕ ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА НА БАЗЕ КОНЕЧНОГО АВТОМАТА

2.1. 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 ЗФО 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.

2.2. Пример построения МДКА для сканера

Пусть исходные данные имеют вид, представленный ниже

Служебные слова:

STA – начало программы.

STO – окончание.

Описание идентификатора:

начинается буквой латинского алфавита, а заканчивается буквой Е.

+, -, *, /.

: =.

Однолитерные разделители:

восьмеричная дробь, например – 617.214

Двулитерные разделители:

Константа:

**Примерный фрагмент программы
для анализа:**

STA

$xE := yE + 1.1$

$y123E := xE * yE - 2.257$

STO

Необходимо построить МДКА, выполняющий функции сканера

2.2.1. 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

2.2.2. Построение регулярного выражения по варианту исходных данных

Точное описание конечного автомата возможно получить на основании общего вида литерных цепочек, которых необходимо идентифицировать, с использованием регулярных выражений. Правила конструирования регулярных выражений изложены в [9, 10, 12].

Общая структура регулярного выражения для описания конечного автомата, дополненного функциями сканера, примерно такова:

$$Cc_i L_1 \vee Cc_2 L_1 \vee <iden> L_1 \vee <data> L_2 \vee d_1 \vee d_2 \vee \dots \vee d_j \vee \dots \vee d_n \vee L_3.$$

В записи обозначено:

Cc_i – i -тое служебное слово.

L_1 – лексема, состоящая в том, что текущая литера не буква и 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

$$B \{ B \vee \mathcal{L} \} E,$$

где B – любая латинская буква, \mathcal{L} – любая десятичная цифра, E – буква, которой, по условию задания, оканчивается имя переменной.

Восьмеричная дробная константа описывается следующим образом:

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

где u – восьмеричная цифра из диапазона 0÷7.

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

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

где δ – буква латинского алфавита, не совпадающая по начертанию с литерами S, T, A, O, E.

2. Множество цифр $\mathcal{L} = \{8, 9, u\}$.

3. Множество «не буква, не цифра» $L_1 = \{+, -, *, /, :, =, \bullet\} \cup L_3$, включает, помимо однолитерных разделителей и компонентов двулитерных разделителей, точку, отделяющую целую часть числа от дробной части, и любой символ, не принадлежащий алфавиту.

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

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

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

Подставим дизъюнктивные члены в общее регулярное выражение, и приступим к построению функции переходов.

2.2.3. Построение функции переходов МДКА

Процесс разметки и минимизации представлен на рисунке 2.1. На фрагменте 2.1, а, показано выражение до минимизации, на 2.1, б, – окончательно минимизированное выражение.

В результирующем выражении видно, что по цепочкам литер, в которых начало идентификатора полностью или частично совпадает со служебным словом, 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]. Однако, как это видно из предыдущего раздела, посвящённого построению МДКА, при этом появляются новые столбцы в таблице функции переходов, а часть столбцов, для которых первоначально проявлялась неоднозначность перехода, становятся недостижимыми. При этом, вновь возникающие образы функции переходов в новых столбцах, частично поглощают образы из недостижимых состояний.

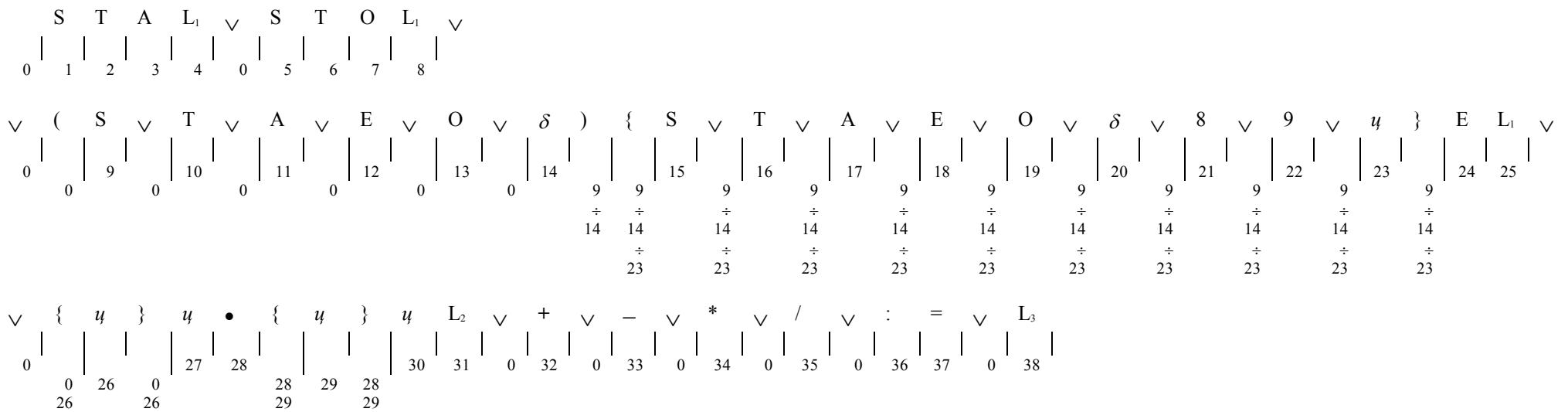
Это наблюдение позволяет провести преобразование НДКА в ДКА, если воспользоваться описанием КА в виде графа и с его помощью построить функцию переходов ДКА. Суть подхода продемонстрируем с помощью рисунка 2.2, на котором изображён фрагмент графа конечного автомата, в части, касающейся служебных слов языка.

Из рисунка видно, что если возникает цепочка, соответствующая служебному слову, то автомат дойдёт до одного из конечных состояний. Если же, на каком-либо такте работы КА, начиная со 2-й литеры, она не совпадёт с соответствующей буквой служебного слова, то, вероятно, анализируемая цепочка принадлежит к классу идентификаторов.

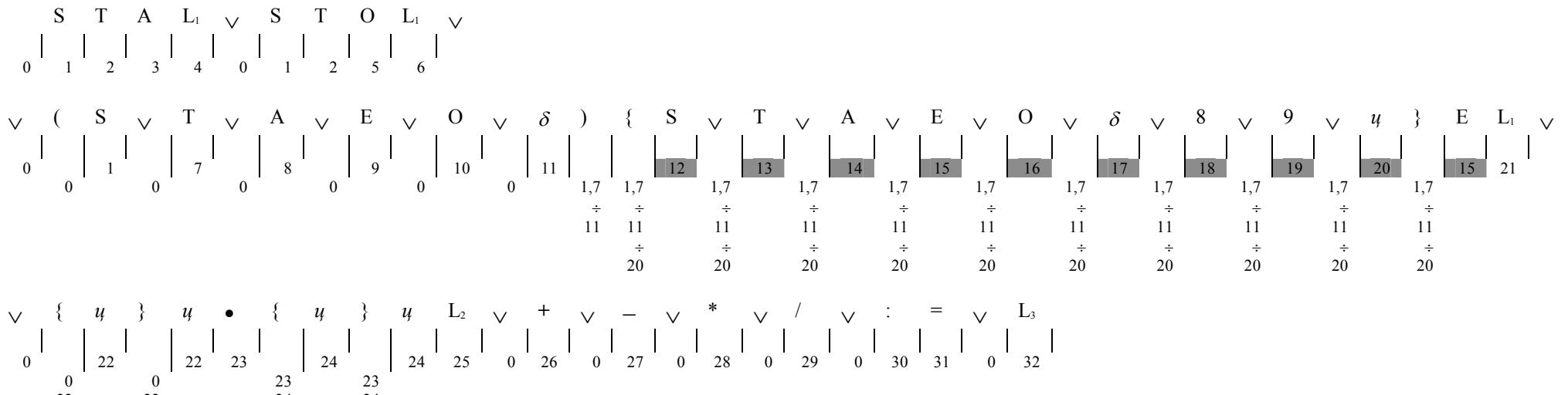
В этом случае, её описание подчиняется третьему дизъюнктивному члену регулярного выражения и, в частности, его части, заключённой в итерационные скобки, что соответствует, буквам и цифрам, следующим за 1-й буквой имени переменной.

Поэтому элементы функции переходов могут быть дополнены (скорее, доопределены) с использованием регулярного подвыражения описания идентификатора.

В таблице 2.1 – представлена функция переходов детерминированного конечного автомата.



a)



б)

Рисунок 2.1 – Минимизация конечного автомата по регулярному выражению

Таблица 2.1 – Функция перехода ДКА, построенная по минимизированному регулярному выражению

	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	12	12	12		12		12	12	12	12	12	12	12	12	12	12	12	12	12	12				25								
T	7	(2)	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				25								
O	9	16	(5)	16		16		16	16	16	16	16	16	16	16	16	16	16	16	16	16				25								
E	10	15	15	15		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				25								
8		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				25								
ц	22	20	20	20		20		20	20	20	20	20	20	20	20	20	20	20	20	20	20				25								
+	26			4		6											21																
-	27			4		6											21																
*	28			4		6											21																
/	29			4		6											21																
:	30			4		6											21																
·				4		6											21																
=				4		6											21																
L3	32			4		6											21																31

Служебное слово STA

Идентификатор

Восьмеричная константа

Знак +

Знак -

Знак *

Знак /

Оператор присваивания

Синтаксическая ошибка

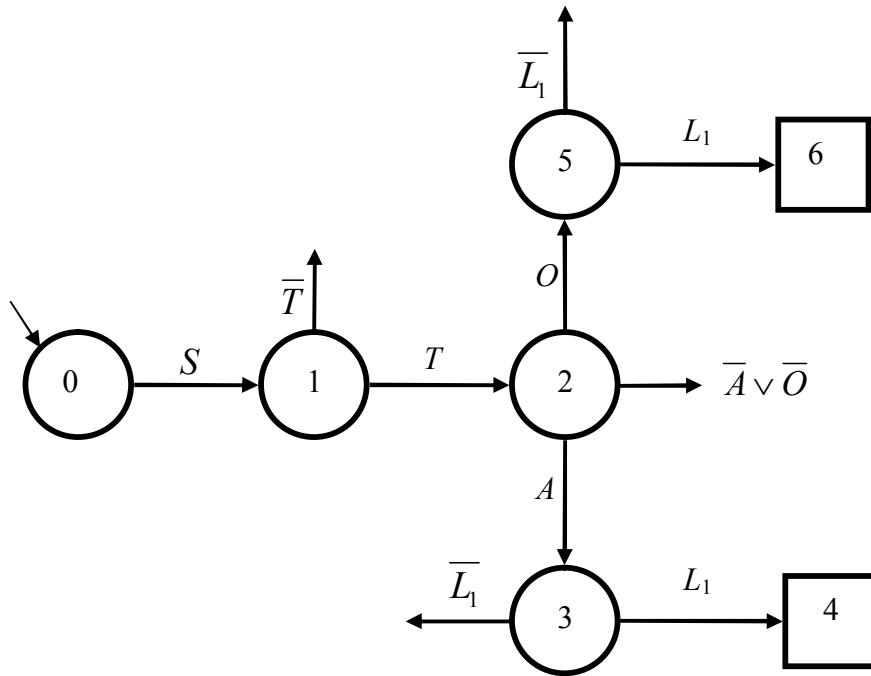


Рисунок 2.2 – Фрагмент КА, относящийся к разбору служебных слов

Фрагмент графа конечного автомата, отображенный на рисунке 2.2, показан в столбцах таблицы 2.1 с номерами 1 и 2 значениями, заключенными в скобки и выделенные фоном. Прочие элементы указанных столбцов записываются, согласно разметке части регулярного выражения, относящегося к описанию идентификатора регулярного выражения, заключенного в фигурные скобки (смотрите рисунок 2.1, б, основные места с 12-го по 20-е, выделены фоном).

Приступим к **минимизации таблицы**, описывающей 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 “Р” для всех состояний, кроме заключительных состояний;
- **сигналы заключительных состояний**, соответствующих опознанным лексемам, это 4, 6, 21, 25 – 29, 31, 32. В каждом из этих состояний генерируется **свой уникальный сигнал**. Его можно считать равным **коду лексемы**.

Заключительные состояния можно удалить из таблицы как неинформативные: хотя образы функции перехода КА в эти состояния присутствуют, переход КА в начальное состояние из заключительных будет осуществляться принудительно, а не под воздействием входной последовательности.

Множества неразличимых состояний, подпадающие под процедуру минимизации, суть {7 – 15, 17 – 20}. Конечные состояния {4, 6, 21, 25 – 29, 31, 32} тоже устраним из таблицы переходов, а соответствующую позицию функции $\delta(q_r, ar)$ определим используя коды, присвоенные этим лексемам на этапе кодирования.

Для осуществления корректной минимизации, в таблице 2.2 приводится система перенумерации состояний для таблицы минимального автомата.

Таблица 2.2. – 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

Для случаев, когда образ функции переходов КА пуст (см. исходную таблицу 2.1), введём коды ошибок, которые в ходе работы сканера помогут генерировать осмысленные диагностические сообщения.

1. Некоректный 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 $L_2 \setminus \{\bullet\}$. Состояние 23, все строки, кроме «ц», код 804.
5. Ошибка в операторе присваивания. Состояние 30, все строки, кроме «==», код 805.

Функция переходов минимального детерминированного конечного автомата приводится в таблице 2.3.

Введём в алфавит входных символов конечного автомата пробел (строка № 16 таблицы 2.3), пополняющий множество L_1 (не буква, не цифра) для упрощения подготовки тестовых данных с позиции их наглядности (читабельности).

Таблица 2.3 – 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
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
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

2.2.4 Методология разработки программы сканера

В общем случае алгоритм работы программы моделирования автомата может выглядеть таким образом. Организация цикла чтения из файла по одной лигере 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 лигере 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).

Особое внимание обратим на то, что при опознавании сканером служебного слова, переменной или константы, номер состояния КА полагается равным начальному, а **чтение очередной личности** из входного файла не выполняется! Это происходит потому, что личность, приводящая к опознанию, на данном шаге квалифицируется как одна из лексем и будет анализироваться на следующем.

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

Фрейм программы представлен ниже, на рисунке 2.3.

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

Отметим, что в учебной программе не показаны 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 личтер приведены в таблице 2.4.

Таблица 2.4 – Правильные и ошибочные цепочки

Цепочка личтер	Результат обработки
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

```

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);

```

Рисунок 2.3 – Структура программы сканера

3. ПОСТРОЕНИЕ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ

3.1. 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 et c. 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.

Покажем на примерах, как 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]. Демонстрация будет выполняться в следующей последовательности.

1. На основании варианта задания (фрагмента программы) разрабатывается формальная грамматика;
2. Продукции грамматики анализируются на свойство $LL(1)$, и, 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.

3.2. Построение нисходящего распознавателя

3.2.1. 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 иерархической 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 быть представлена единственным оператором, то есть

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

Либо, используя прямую правую рекурсию, оператором, за которым следует последовательность операторов

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

В соответствии с ранее принятой программой составления грамматики, 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, например,

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

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

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

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

и знаков операций:

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

Окончательно грамматика будет выглядеть следующим образом:

$\langle \text{фрагмент} \rangle ::= \text{STA } \langle \text{операторы} \rangle \text{ STO}$
 $\langle \text{операторы} \rangle ::= \langle \text{оператор} \rangle | \langle \text{оператор} \rangle \langle \text{операторы} \rangle$
 $\langle \text{оператор} \rangle ::= \langle \text{idен} \rangle := \langle \text{выражение} \rangle,$

```

<выражение> ::= <операнд>|<операнд><знак><выражение>
<операнд> ::= <иден>|<дата>
<знак> ::= +|-|*|/

```

Понятия грамматики, такие как **идентификатор** (**<иден>**) и **константа** (**<дата>**) обычно описываются в форме Бэкуса-Наура, хотя считаются условно терминальными символами, так как их опознание и контроль правильности написания осуществляются на этапе лексического анализа. Мы же это описание опустим, как несущественное при решении нашей задачи.

Элементами разработанной грамматики являются: 1) словарь нетерминалов $V_N = \{\text{фрагмент, операторы, оператор, выражение, operand, знак}\}$; 2) словарь терминалов $V_T = \{\text{STA, STO, +, -, *, /, :=, <iden>, <data>}\}$; 3) аксиома грамматики – нетерминальный символ **<фрагмент>**; множество продукции – правила подстановки (1) – (12).

Для контроля корректности построенной нами формальной грамматики, то есть релевантности продукции и непустоты языка, построим синтаксическое дерево, приведенное на рисунке 3.1, хотя исследование этих свойств проводят намного скрупулённее [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 продукции грамматики, для продукции с одинаковыми **левыми частями**, **не пересекались**. Собственно, процедура исследований изложена в [3, 11], проиллюстрирована в [9], результаты исследования грамматики помещены в таблицу 3.1. 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 3.1 – Исследование $LL(1)$ – свойств формальной грамматики

№	Продукции грамматики	Цепочки для $k=1$
1	<фрагмент> → STA<операторы>STO	{STA}
2	<операторы> → <оператор>	{<иден>}
3	<операторы> → <оператор><операторы>	{<иден>}
4	<оператор> → <иден>:=<выражение>	{<иден>}
5	<выражение> → <операнд>	{<иден>, <дата>}
6	<выражение> → <операнд><знак><выражение>	{<иден>, <дата>}
7	<операнд> → <иден>	{<иден>}
8	<операнд> → <дата>	{<дата>}
9	<знак> → +	{+}
10	<знак> → -	{-}
11	<знак> → *	{*}
12	<знак> → /	{/}

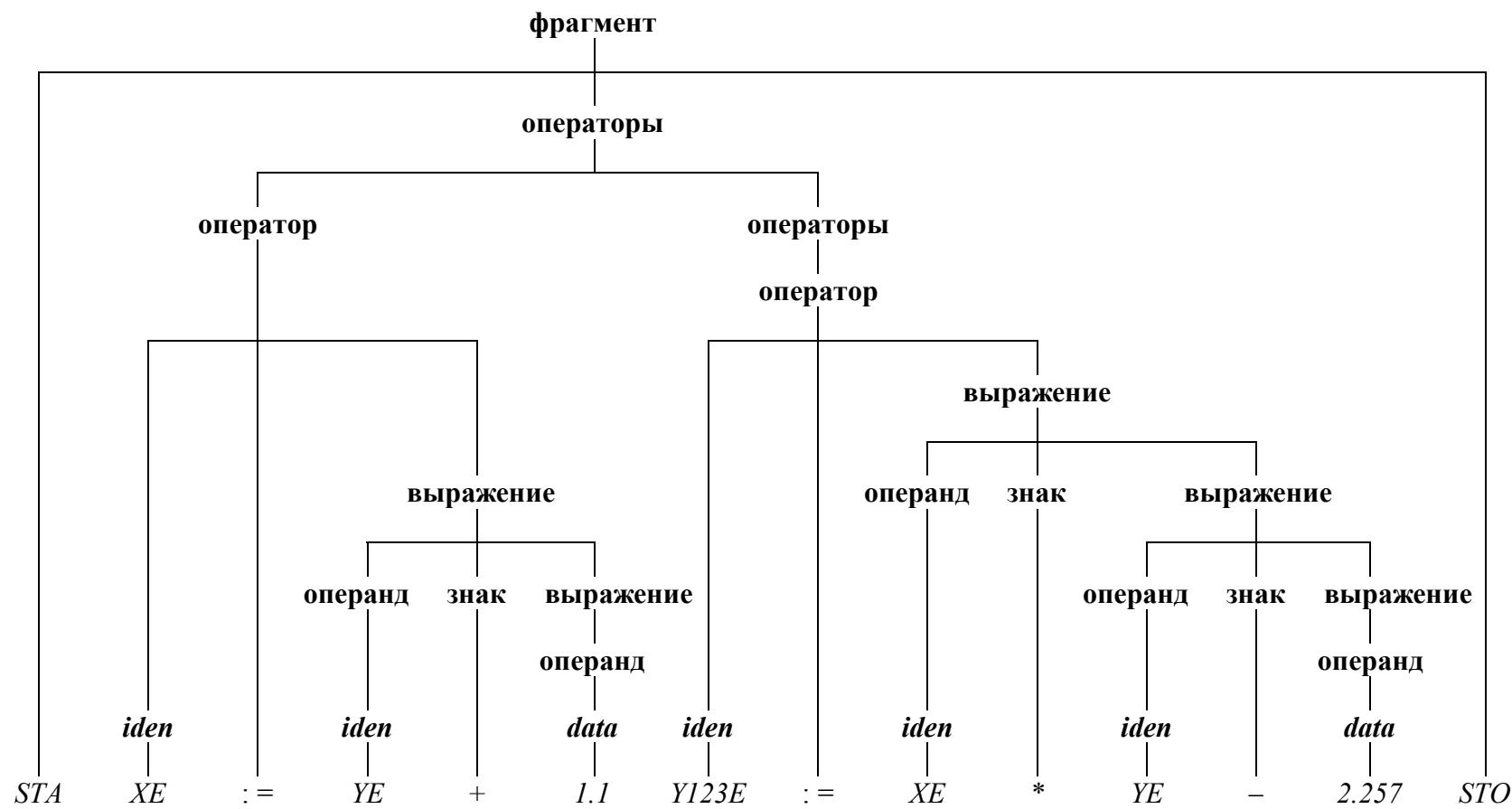


Рисунок 3.1 – Синтаксическое дерево фрагмента учебной программы

Свойство $LL(1)$ нарушается для правил (2) и (3), описывающих нетерминальный символ <операторы>, и правил (5) и (6) для нетерминала <выражение> (выделено в таблице 3.1 серым цветом). 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:

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

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

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

$$<\text{фрагмент}> ::= STA <\text{тело программы}> \quad (15)$$

Нетерминал <тело программы> определим как

$$<\text{тело программы}> ::= <\text{оператор}> <\text{тело программы}> \quad (16)$$

$$<\text{тело программы}> ::= STO \quad (17)$$

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

$$<\text{выражение}> ::= <\text{операнд}> <\text{продолжение}> \quad (18)$$

$$<\text{продолжение}> ::= <\text{знак}> <\text{продолжение}> \quad (19)$$

$$<\text{продолжение}> ::= <\text{операнд}> \quad (20)$$

$$<\text{продолжение}> ::= \epsilon \quad (21)$$

В продукции (21) ϵ обозначен пустой символ (цепочка). ϵ - 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.

$$<\text{фрагмент}> ::= STA <\text{тело программы}> \quad (22)$$

$$<\text{тело программы}> ::= <\text{оператор}> <\text{тело программы}> \quad (23)$$

$$<\text{тело программы}> ::= STO \quad (24)$$

$$<\text{оператор}> ::= <\text{id}> ::= <\text{операнд}> \{<\text{знак}> <\text{операнд}>\}^{\infty}_0 \quad (25)$$

$$<\text{операнд}> ::= <\text{id}> \quad (26)$$

<операнд> ::= <data> (27)

<знак> ::= + (28)

<знак> ::= — (29)

<знак> ::= * (30)

<знак> ::= / (31)

Таким образом, в полученной $LL(1)$ -грамматике десять продукции (22) – (31) наряду со следующими элементами:

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

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

аксиома грамматики ФРАГМЕНТ.

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

3.2.2 Реализации нисходящего распознавателя методом рекурсивного спуска

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 увязывать с фактом неверного опознания того или иного символа.

В программе, соответствующей грамматики (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 лексемы*/
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(Prg_code, "%d", &code);
                      return 1;}
    else { printf ("\n Нет знака присваивания");
           return 0;}
}

```

Примеры правильной и ошибочных тестовых последовательностей приводятся в таблице 2.3.

Таблица 3.2 – 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.

3.3. Построение восходящего распознавателя

Чтобы воспользоваться алгоритмом разбора, основанным 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 продукции грамматики. Показаниями для этой операции служит наличие нетерминального символа внутри цепочки правой части правила, который затем описывается с использованием левой или правой рекурсии [7].

3.3.1. Исследование грамматики на предшествование

В качестве исходного материала для исследования возьмем грамматику с продукциями (1) – (12), ранее нами составленную. Нетерминальный символ *<операторы>*, стоящий посреди правила (1), праворекурсивно описывается правилом (3). Таким образом, между парой символов грамматики *<операторы>...STO* будут иметь место сразу два отношения *<операторы> • > STO* и *<операторы> ≡ STO*, что ясно видно на синтаксическом дереве (рисунок 3.1). 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

<фрагмент> ::= STA <последовательность> STO (32)

<последовательность> ::= <операторы> (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, но мера сия является вынужденной.

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:

<фрагмент> ::= STA <тело программы>

<тело программы> ::= <оператор> <тело программы> | STO

<оператор> ::= <iden> := <выражение>

<выражение> ::= <операнд> | <операнд> <знак> <выражение>

<операнд> ::= <iden> | <data>

*<знак> ::= + | - | * | /*

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 $V_T = \{\text{STA}, \text{STO}, \text{iden}, \text{data}, \text{:=}, +, -, *, /\}$ – 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): $EQ(R, S) = 1$, при $E \rightarrow uRSv$, где $E \in V_N$, $R, S \in V$, $u, v \in V^*$.

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

3. L (last): $L(<R>, S) = 1$, при $R \rightarrow Su$, где $R \in V_N$, $S \in V$, $v \in V^*$.

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 3.3. Буквами e , f , l обозначены соответствующие значения.

Таблица 3.3 – Совмещенная карта отношений F, L, EQ

Заменяя буквы единицами, трансформируем таблицу в двоичные матрицы отношений:

$$l = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

На основании матриц F , L вычисляются транзитивные замыкания отношений F^+ и L^+ .

Транзитивное замыкание отношения заданного булевой матрицей A вычисляется по правилу

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

Причем возвведение 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'e приводится ниже.

```
Procedure Warshall;
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, каждой 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 3.4 – Отношение предшествования для разработанной грамматики

$\begin{array}{c} S \\ \diagdown \\ R \end{array}$	фрагмент	Тело про- граммы	Оператор	выраж- ение	операнд	знак	STA	STO	iden	data	$:=$	$+$	$-$	$*$	$/$
фрагмент															
Тело про- граммы															
Оператор	≡	$< \bullet$					$< \bullet$	$< \bullet$							
выражение	$\bullet >$	$\bullet >$					$\bullet >$	$\bullet >$							
операнд	$\bullet >$	$\bullet >$			≡		$\bullet >$	$\bullet >$			$< \bullet$	$< \bullet$	$< \bullet$	$< \bullet$	
знак			≡	$< \bullet$					$< \bullet$	$< \bullet$					
STA	≡	$< \bullet$					$< \bullet$	$< \bullet$							
STO															
iden						$\bullet >$				≡	$\bullet >$	$\bullet >$	$\bullet >$	$\bullet >$	
data						$\bullet >$					$\bullet >$	$\bullet >$	$\bullet >$	$\bullet >$	
$:=$		≡	$< \bullet$						$< \bullet$	$< \bullet$					
$+$			$\bullet >$	$\bullet >$					$\bullet >$	$\bullet >$					
$-$			$\bullet >$	$\bullet >$					$\bullet >$	$\bullet >$					
$*$			$\bullet >$	$\bullet >$					$\bullet >$	$\bullet >$					
$-$			$\bullet >$	$\bullet >$					$\bullet >$	$\bullet >$					

3.3.2. Программная реализация 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 – operand, 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 и оба этих символов являются терминалами. В данном случае выдается диагностическое сообщение вида “Символы R и S не должны стоять рядом”.
- Второй случай возникает тогда, когда символы R и/или S нетерминалы, они представляют собой результат подстановок, выполненных на предыдущих шагах. В этом случае, сообщение о неправильном соседстве может обескуражить, поэтому диагностическое сообщение должно включать понятийные термины, как-то: “Заголовок цикла оформлен неправильно”, “Некорректное оформление условного оператора” и т.п.

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 3.5 – 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	4
4	2	1	6	6	6	4	1	1	6	6	6	6	6	6	6
4	3	3	6	6	6	3	3	6	6	6	6	6	6	6	6
4	3	3	6	6	2	6	3	3	6	6	1	1	1	1	1
4	6	6	2	1	6	6	6	1	1	6	6	6	6	6	6
4	2	1	5	5	5	5	1	1	5	5	5	5	5	5	5
4	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	3
4	6	6	6	6	1	4	4	6	6	6	3	3	3	3	3
4	7	7	2	1	7	4	4	1	1	7	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	7	7	7	7	7
4	7	7	3	3	7	4	4	3	3	7	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, 6, 1},
/*3*/ {4, 3, 3, 6, 6, 6, 6, 3, 3, 6, 6, 6, 6, 6, 6, 6, 1},
/*4*/ {4, 3, 3, 6, 6, 2, 6, 3, 3, 6, 6, 1, 1, 1, 1, 1, 1},
/*5*/ {4, 6, 6, 2, 1, 6, 6, 6, 1, 1, 6, 6, 6, 6, 6, 6, 1},
/*6*/ {4, 2, 1, 5, 5, 5, 5, 1, 1, 5, 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, 7, 1},
/*10*/ {4, 7, 7, 2, 1, 7, 4, 4, 1, 1, 7, 7, 7, 7, 7, 7, 1},
/*12*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 7, 1},
/*13*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 7, 1},
/*14*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 7, 1},
/*15*/ {3, 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 3.6.

Таблица 3.6 – 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)

ЗАКЛЮЧЕНИЕ

Освещённые в настоящем методическом указании вопросы ни в коей мере не претендуют на всемерный охват области информационных технологий, посвящённых синтаксическому анализу. Желающие, по мере необходимости, могут пополнять и углублять свои знания, пользуясь специальной литературой. Выражаем надежду и уверенность, что данная методическая разработка не только послужит хорошим начальным подспорьем для дальнейшего освоения дисциплины, но и поможет сделать первые шаги, которые, как известно, порой нелегки...

С уважением
В.Ю. Карлусов
С.А. Кузнецов

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. Улман. – М. : Издательский дом «Вильямс», 2002. – 528 с.

Электронные издания

13. Малявко, А. А. Формальные языки и компиляторы : учеб. пособие для вузов / А. А. Малявко. — Москва : Издательство Юрайт, 2019. — 429 с. — (Серия : Университеты России). — ISBN 978-5-534-04288-7. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: <https://biblio-online.ru/book/formalnye-yazyki-i-kompilyatory-438060>.

14. Кудрявцев, В. Б. Теория автоматов : учебник для бакалавриата и магистратуры / В. Б. Кудрявцев, С. В. Алешин, А. С. Подколзин. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 320 с. — (Серия : Бакалавр и магистр. Академический курс). — ISBN 978-5-534-00117-4. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: <https://biblio-online.ru/book/teoriya-avtomatov-444091>.

15. Введение в теорию алгоритмических языков и компиляторов: учеб. пособие / Л. Г. Гагарина, Е.В. Кокорева. - М. : ИД ФОРУМ, 2011. - 176 с.: ил.; 60x90 1/16. - (Высшее образование). (переплет) ISBN 978-5-8199-0404-6 - Режим доступа: <http://znanium.com/catalog/product/265617>

16. Алымова, Е. В. Конечные автоматы и формальные языки : учебник / Е. В. Алымова. В. М. Деундяк. А. М. Пеленцын ; Южный федеральный университет. - Ростов-на-Дону : Таганрог : Издательство Южного федерального университета. 2018. - 292 с. - ISBN 978-5-9275-2397-9. - Режим доступа: <http://znanium.com/catalog/product/1020503>

ПРИЛОЖЕНИЕ А
(обязательное)

Варианты, по усмотрению преподавателя, могут быть выданы по номерам зачётной книжки с использованием таблицы А.1.

Таблица А.1 – Список вариантов задания к построению МДКА

1-я цифра	2-я цифра									
	1	2	3	4	5	6	7	8	9	0
1	A \vee B \vee C	D \vee EF	GH \vee I	J \vee K \vee L	KL \vee A	E \vee A \vee L	B \vee KC	JD \vee I	F \vee H \vee G	G \vee EA
2	B \vee CD	E \vee F \vee G	HI \vee J	H \vee EB	I \vee A \vee L	LB \vee K	C \vee J \vee D	I \vee FH	GH \vee K	B \vee E \vee I
3	J \vee FC	DG \vee K	I \vee A \vee J	F \vee GH	BD \vee E	L \vee CF	JA \vee D	F \vee G \vee E	L \vee FG	EH \vee D
4	A \vee D \vee G	B \vee IK	CL \vee H	F \vee J \vee E	G \vee BI	I \vee C \vee J	B \vee KA	AL \vee F	G \vee E \vee H	D \vee IC
5	KC \vee L	H \vee FJ	E \vee A \vee D	L \vee HF	JE \vee A	JB \vee K	J \vee E \vee K	A \vee D \vee F	H \vee BI	LC \vee G
6	D \vee G \vee B	I \vee KC	AL \vee J	F \vee C \vee D	G \vee KH	G \vee J \vee E	K \vee AD	FH \vee B	I \vee L \vee C	C \vee GJ
7	EB \vee I	C \vee D \vee E	D \vee IF	IA \vee L	J \vee F \vee C	BA \vee D	I \vee C \vee J	E \vee FK	GL \vee H	H \vee B \vee A
8	D \vee GK	AL \vee B	K \vee C \vee J	H \vee GE	HK \vee B	D \vee IC	JE \vee F	K \vee G \vee L	E \vee KA	DF \vee H
9	E \vee I \vee L	C \vee FJ	AD \vee G	D \vee G \vee H	K \vee BE	B \vee I \vee L	L \vee HB	AD \vee I	C \vee J \vee E	F \vee KG
0	IL \vee C	F \vee J \vee A	H \vee DI	CJ \vee B	K \vee A \vee L	BE \vee D	G \vee L \vee H	C \vee IL	JA \vee H	I \vee C \vee A

В таблице А.1 приняты следующие обозначения:

A	-	$ab\{b \vee a\}$	G	-	$b\{a \vee ac\}$
B	-	$\{ab\}c\{c\}$	H	-	$c\{bb \vee ba\}$
C	-	$ca\{b\}$	I	-	$b\{b\}a$
D	-	$b\{c \vee a\}$	J	-	$a\{a \vee ab\}$
E	-	$\{a \vee c\}b$	K	-	$b\{ab\}c\{a\}$
F	-	$c\{a \vee b\}$	L	-	$b\{ba \vee bb \vee bc\}$.

ПРИЛОЖЕНИЕ Б
(обязательное)

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

Таблица Б.1 – Варианты задания для процедур лексического и 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 Б.2 – Требования к оформлению идентификаторов

№	Описание идентификатора
(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

Таблица Б.3 – Типы констант

№	Тип	Пояснения	Формат
1	F	С фиксированной точкой	$\pm d \dots d.d \dots d$
2	E	С плавающей точкой	$\pm d \dots d.d \dots dE \pm dd$
3	B	Двоичная	$\pm 2 \dots 2B$
4	C	Символьная	's ... s'
5	X	Шестнадцатеричная	$\pm 0xh \dots h$
6	O	Восьмеричная	$\pm 0o8 \dots 8$
7	D	Удвоенная, с плавающей точкой	$\pm d \dots d.d \dots dD \pm dd$
8	L	Удвоенная, целая	$\pm dd \dots ddL$

Примечания.

- d – десятичная цифра.
- h – шестнадцатеричная цифра.
- 8 – восьмеричная цифра.
- 2 – двоичная цифра.
- s – любой символ.
- 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.

Важное замечание.

Вид идентификаторов и констант, приводимых в графе (5) ниже следующей таблице, как правило, не соответствует доставшимся Вам по варианту. Фрагмент служит лишь для иллюстрации синтаксиса.

Таблица Б.4 – Фрагменты учебных программ для анализаторов

№	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;

(1)	(2)	(3)	(4)	(5)
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)
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; }

(1)	(2)	(3)	(4)	(5)
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) \$
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:

(1)	(2)	(3)	(4)	(5)
15	WAIT SIGNAL	() > = + %	>= <=	WAIT(S) K=K+5.3 % C=C+1 % SIGNAL(S<=0xf)