1 Тема и цель курсовой работы

Тема курсовой работы: «Разработка распознавателя модельного языка программирования».

Цель курсовой работы:

- закрепление теоретических знаний в области теории формальных языков, грамматик и автоматов;
- формирование практических умений и навыков разработки собственного распознавателя модельного языка программирования;
- закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

2 Основы теории автоматов и формальных языков

2.1 Описание синтаксиса языка программирования

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта.

Формальные грамматики

Определение 2.1. Формальной грамматикой называется четверка вида:

$$G = (V_T, V_N, P, S), \tag{1.1}$$

где V_N - конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы);

V_T - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.), $V_T \cap V_N = \varnothing$;

P – множество правил вывода грамматики, являющееся конечным подмножеством множества $(V_T \cup V_N)^+ \times (V_T \cup V_N)^*$; элемент (α, β) множества P называется правилом вывода и записывается в виде $\alpha \to \beta$ (читается: «из цепочки α выводится цепочка β»);

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

Для записи правил вывода с одинаковыми левыми частями вида $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$ используется сокращенная форма записи $\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$.

Пример 2.1. Опишем с помощью формальных грамматик синтаксис паскалеподобного модельного языка *M*. Грамматика будет иметь правила вывода вида:

PR→{*BODY*}

BODY→*DESCR* | *OPER* | *BODY*; *OPER* | *BODY*; *DESCR*

DESCR→% ID_1 | # ID_1 | $ ID_1

ID_1→ *ID* | ID_1, *ID*

$OPER_1$→ *OPER* | $OPER_1$: *OPER*

OPER →[$OPER_1$] | *if COMPARE then OPER* |

if COMPARE then OPER else OPER | *while COMPARE do OPER* |

for ID as COMPARE to COMPARE do OPER | *read*(ID_1) |

write(*EXPR*) | *ID as COMPARE*

EXPR→ *COMPARE* | *EXPR*, *COMPARE*

COMPARE→*ADD* | *COMPARE*=*ADD* | *COMPARE*>*ADD* |

COMPARE<*ADD* | *COMPARE*>=*ADD* | *COMPARE*<=*ADD* |

COMPARE<>*ADD*

ADD→ *MULT* | *ADD* + *MULT* | *ADD* - *MULT* | *ADD or MULT*

MULT→ *FACT* | *MULT*/*FACT* | *MULT***FACT* | *MULT and FACT*

FACT→*ID* | *NUM* | *LOG* | *not FACT* | (*COMPARE*)

LOG→*true* | *false*

ID→*CH* | *ID CH* | *ID DIGIT*

$DIGIT_1 \rightarrow DIGIT \mid DIGIT_1\ DIGIT$

$DIGIT \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$CH \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \mid$
$A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid$
$V \mid W \mid X \mid Y \mid Z$

$NUM \rightarrow INT \mid REAL$

$INT \rightarrow BIN \mid OCT \mid DEC \mid HEX$

$BIN \rightarrow BIN_1\ B \mid BIN_1\ b \mid BIN_1\ BIN$

$BIN_1 \rightarrow 0 \mid 1$

$OCT \rightarrow OCT_1\ O \mid OCT_1\ o \mid OCT_1\ OCT$

$OCT_1 \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

$DEC \rightarrow DIGIT_1\ D \mid DIGIT_1\ d \mid DIGIT_1$

$HEX \rightarrow HEX_1\ H \mid HEX_1\ h$

$HEX_1 \rightarrow DIGIT \mid HEX_1\ DIGIT \mid HEX_1\ CH_1$

$CH_1 \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid A \mid B \mid C \mid D \mid E \mid F$

$REAL \rightarrow DIGIT_1\ POR \mid DIGIT_1.DIGIT_1\ POR \mid .DIGIT_1\ POR \mid .DIGIT_1 \mid$
$DIGIT_1.DIGIT_1$

$POR \rightarrow E+DIGIT_1 \mid E\text{-}DIGIT_1 \mid e+DIGIT_1 \mid e\text{-}DIGIT_1 \mid E\ DIGIT_1 \mid e\ DIGIT_1$

Формы Бэкуса-Наура (БНФ)

Метаязык, предложенный Бэкусом и Науром, использует следующие обозначения:

- символ «::=» отделяет левую часть правила от правой (читается: «определяется как»);
- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;
- терминалы - это символы, используемые в описываемом языке;

- правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или»).

Пример 2.2. Определение понятия «идентификатор» с использованием БНФ имеет вид:

<идентификатор> ::= <буква> | <идентификатор> <буква>
| <идентификатор> <цифра>
<буква> :: = *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z*
<цифра> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Расширенные формы Бэкуса-Наура (РБНФ)

Для повышения удобства и компактности описаний в РБНФ вводятся следующие дополнительные конструкции (метасимволы):

- квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;
- сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;
- круглые скобки «(» и «)» используются для ограничения альтернативных конструкций.

Пример 2.3. В соответствии с данными правилами синтаксис модельного языка *M* будет выглядеть следующим образом:

<буква> :: = *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z*

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<идентификатор> ::= <буква> { <буква> | <цифра> }

<число> ::= {/< цифра> /}

<ключевое_слово> ::= *read* | *write* | *if* | *then* | *else* | *for* | *to* | *while* | *do* | *true* | *false* | *or* | *and* | *not* | *as*

<разделитель> ::= { | } | % | ! | $ | , | ; | [|] | : | (|) | + | - | * | / | = | <> | < | <= | > | >= | /* | */

<программа> ::= «{» <тело> «}»

<тело> ::= (<описание> | <оператор>) {; (<описание> | <оператор>)}

<описание> ::= <тип> <идентификатор> { , <идентификатор> }

<тип>::= % | # | $

<оператор> ::= <присваивания> | <условный> | <фиксированного_цикла> | <условного_цикла> | <составной> | <ввода> | <вывода>

<присваивания> ::= <идентификатор> *as* <выражение>

<условный> ::= *if* <выражение> *then* <оператор> [*else* <оператор>]

<фиксированного_цикла>::= *for* <присваивания> *to* <выражение> *do* <оператор>

<условного_цикла>::= *while* <выражение> *do* <оператор>

<составной>:: = «[» <оператор> { : <оператор> } «]»

<ввода>:: = *read* «(»<идентификатор> {, <идентификатор> } «)»

<вывода>:: = *write* «(»<выражение> {, <выражение> } «)»

<выражение>:: = <сумма> | <выражение> (< > | = | < | <= | > | >=) <сумма>

<сумма> ::= <произведение> { (+ | - | *or*) <произведение>}

<произведение>:: = <множитель> { (* | / | *and*) <множитель>}

<множитель>:: = <идентификатор> | <число> | <логическая_константа> | *not* <множитель> | «(»<выражение>«)»

<логическая_константа>:: = *true* | *false*

<целое>::= <двоичное> | <восьмеричное> | <десятичное> | <шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (*B* | *b*)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (*O* | *o*)

<десятичное>::= {/ <цифра> /} [*D* | *d*]

<шестнадцатеричное>::= <цифра> {<цифра> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* | *c* | *d* | *e* | *f*} (*H* | *h*)

<действительное>::= <числовая_строка> <порядок> | [<числовая_строка>] . <числовая_строка> [порядок]

<числовая_строка>::= {/ <цифра> /}

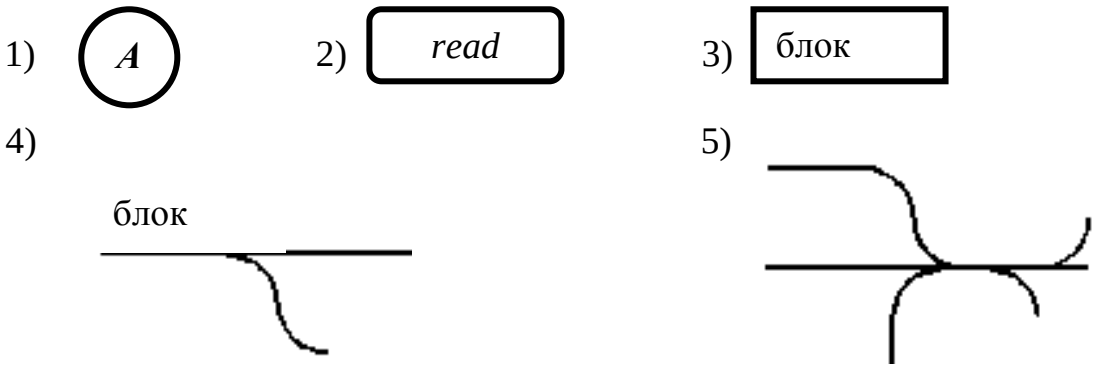
<порядок>::= (E | e)[+ | -] <числовая_строка>

Диаграммы Вирта

В метаязыке диаграмм Вирта используются графические примитивы, представленные на рисунке 2.1.

При построении диаграмм учитывают следующие правила:

- каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно изображаются на противоположных сторонах;
- каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;
- альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;
- должна быть одна входная дуга (располагается обычно слева или сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу);
- стрелки на дугах диаграмм обычно не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений.



1) – терминальный символ, принадлежащий алфавиту языка;

2) – постоянная группа терминальных символов, определяющая название лексемы, ключевое слово и т.д.;

3) – нетерминальный символ, определяющий название правила;

4) – входная дуга с именем правила, определяющая его название;

5) – соединительные линии, обеспечивающие связь между терминальными и нетерминальными символами в правилах.

Рисунок 2.1 – Графические примитивы диаграмм Вирта

Описание синтаксиса модельного языка *M* с помощью диаграмм Вирта представлено на рисунке 2.2.

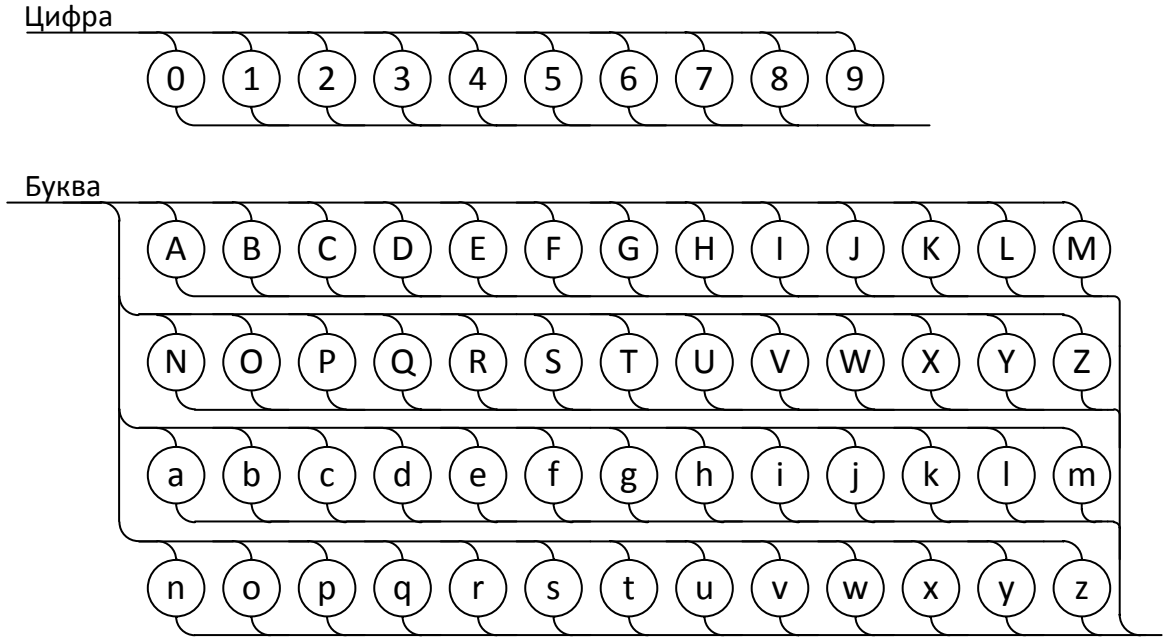


Рисунок 2.2 – Синтаксические правила модельного языка *M*

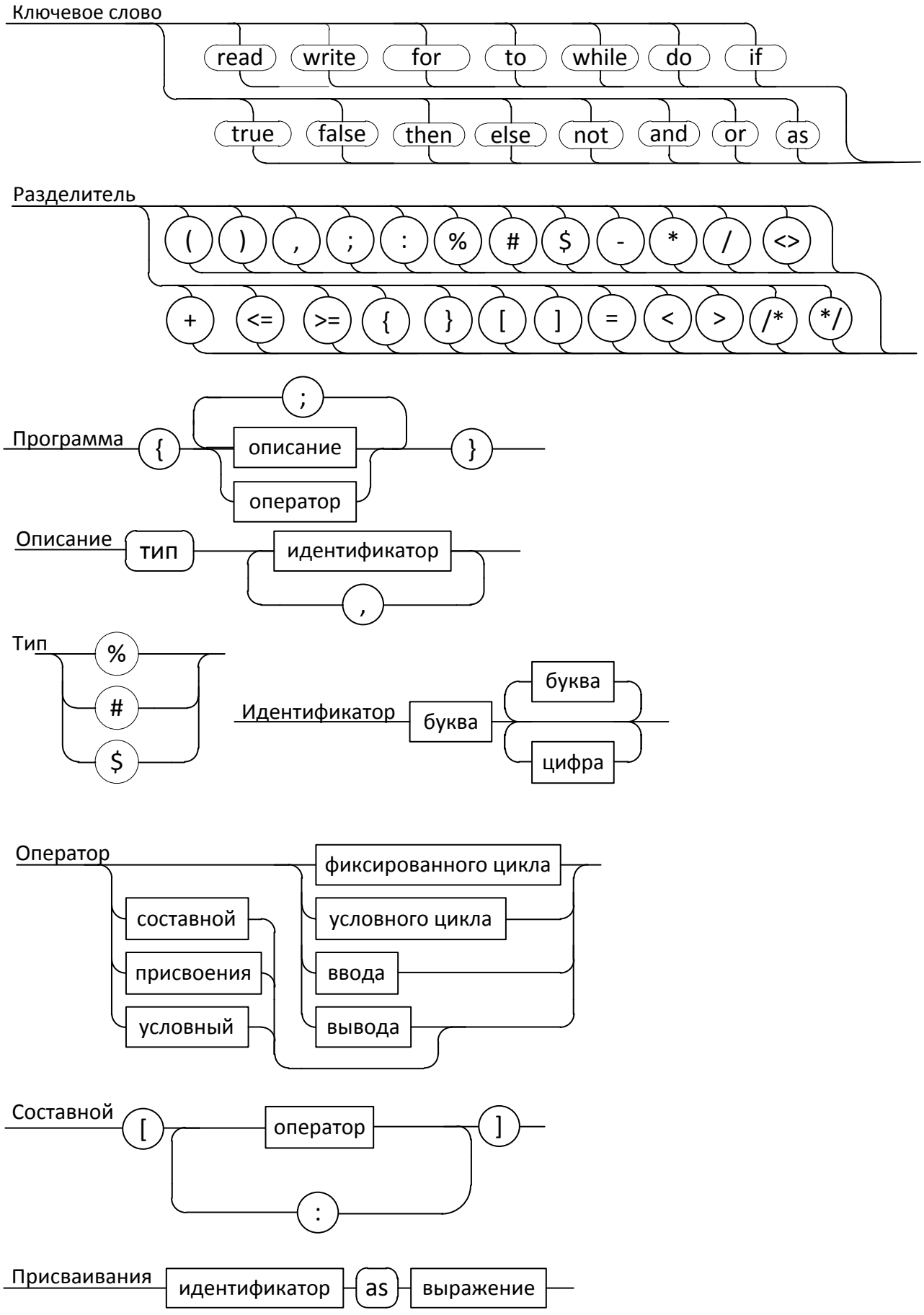


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 2

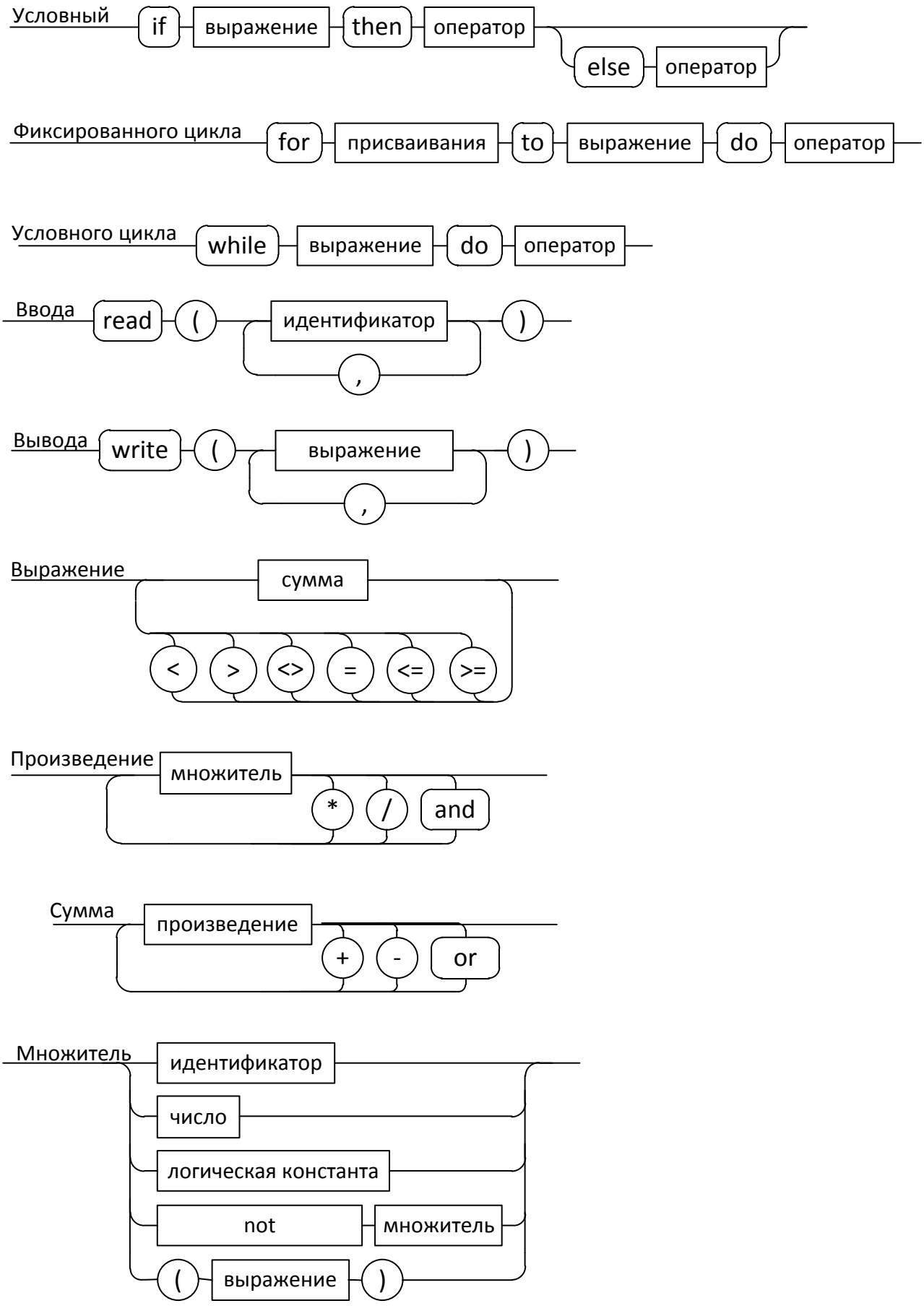


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 3

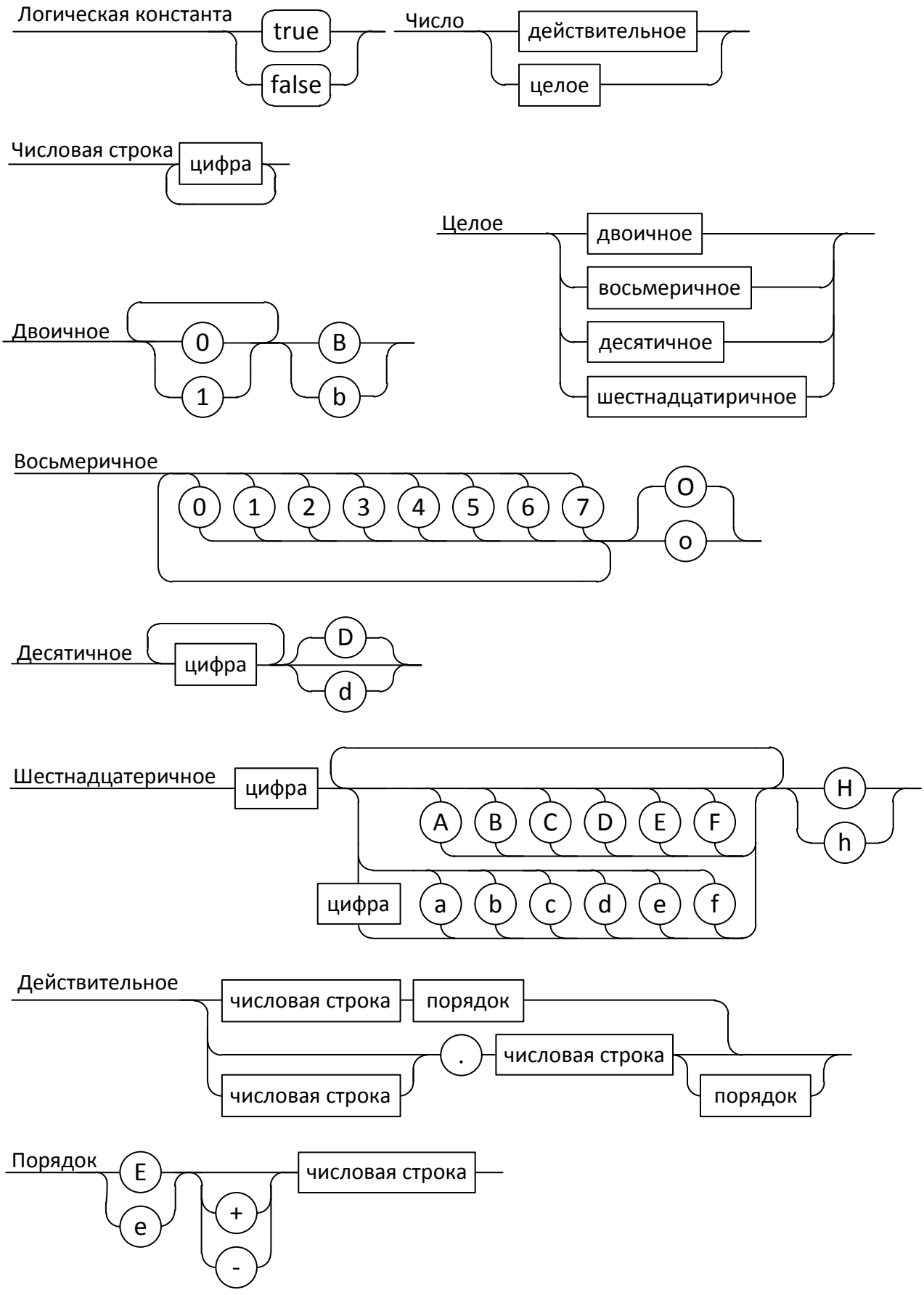


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 4

Пример 2.4. Программа на модельном языке *M*, вычисляющая среднее арифметическое чисел, введенных с клавиатуры.

```
{% i, k, n, sum;
   read(n); sum as 0;
   i as 1;
   while i<=n do [read(k) : sum as sum + k];
   write(sum/n)}
```

2.2 Общая схема работы распознавателя

Определение 2.2. Распознаватель – это специальный алгоритм, который позволяет определить принадлежность цепочки символов некоторому языку.

Распознаватель схематично можно представить в виде совокупности входной ленты, управляющего устройства и вспомогательной памяти (рисунок 2.3).

Входная лента представляет собой последовательность ячеек, каждая из которых содержит один символ некоторого конечного алфавита.

Входная головка в каждый момент обозревает одну ячейку. За один шаг работы распознавателя головка сдвигается на одну ячейку влево (вправо) или остается неподвижной. Распознаватель, не перемещающий входную головку влево, называется односторонним.

Управляющее устройство – это программа управления распознавателем. Она задает конечное множество состояний распознавателя и определяет переходы из состояния в состояние в зависимости от прочитанного символа входной ленты и содержимого вспомогательной памяти.

Вспомогательная память служит для хранения информации, которая зависит от состояния распознавателя. У некоторых типов распознавателей она может отсутствовать.

Поведение распознавателя можно представить с помощью его конфигураций.

Определение 2.3. Конфигурация распознавателя есть совокупность трех элементов:

- состояния управляющего устройства;
- содержимого входной ленты и положения входной головки;
- содержимого вспомогательной памяти.

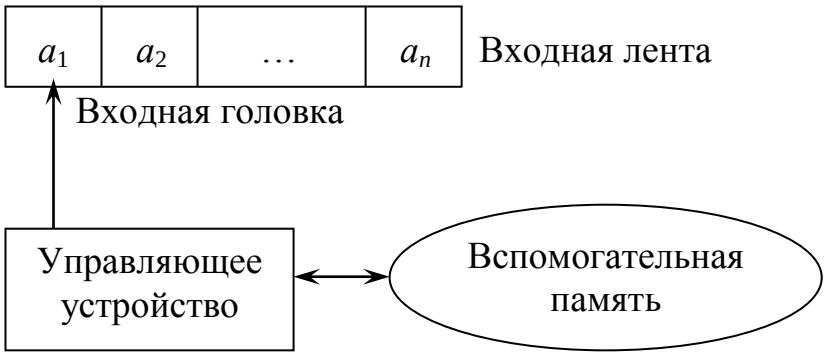


Рисунок 2.3 – Схема распознавателя

Определение 2.4. Управляющее устройство называется детерминированным, если для каждой конфигурации распознавателя существует не более одного возможного следующего шага, в противном случае управляющее устройство называется недетерминированным.

Определение 2.5. Конфигурация распознавателя называется начальной, если управляющее устройство находится в заданном начальном состоянии, входная головка обозревает самый левый символ на входной ленте и вспомогательная память имеет заранее установленное начальное содержимое.

Определение 2.6. Конфигурация распознавателя называется заключительной, если управляющее устройство находится в одном из заранее выделенных заключительных состояний, а входная головка сошла с правого конца входной ленты. Содержимое вспомогательной памяти удовлетворяет некоторому заранее установленному условию.

Определение 2.7. Входная строка ω допускается распознавателем, если от начальной конфигурации, в которой цепочка ω записана на входной ленте, распознаватель может выполнить последовательность шагов, заканчивающуюся заключительной конфигурацией.

Определение 2.8. Множество всех строк, допускаемых распознавателем, называется языком распознавателя.

Классификация распознавателей

Для каждого класса грамматик из иерархии Хомского существует класс распознавателей, определяющих тот же класс языков. Чем шире класс грамматик, тем сложнее класс соответствующих распознавателей.

Для языков типа 0 распознавателями являются машины Тьюринга.

Распознаватели КЗ-языков называются линейными ограниченными автоматами (машины Тьюринга с конечным объемом ленты).

Распознавателями для КС-языков являются автоматы с магазинной памятью (МП-автоматы).

Для регулярных языков распознавателями служат конечные автоматы (КА).

2.3 Лексический анализатор программы

Определение 2.9. Лексический анализатор (ЛА) – это распознаватель, который группируют символы, составляющие исходную программу, в отдельные минимальные единицы текста, несущие смысловую нагрузку – лексемы.

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки. ЛА использует регулярные грамматики.

ЛА является необязательным этапом трансляции, но желательным по следующим причинам:

1) замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки;

2) ЛА уменьшает длину программы, устраняя из ее исходного представления несущественные пробелы и комментарии;

3) если будет изменена кодировка в исходном представлении программы, то это отразится только на ЛА.

В языках программирования лексемы обычно делятся на классы:

1) служебные слова;

2) ограничители;

3) числа;

4) идентификаторы.

Каждая лексема представляет собой пару чисел вида (*n, k*), где *n* – номер таблицы лексем, *k* - номер лексемы в таблице.

Входные данные ЛА - текст транслируемой программы на входном языке.

Выходные данные ЛА - файл лексем в числовом представлении.

Пример 2.5. Для модельного языка *M* таблица служебных слов будет иметь вид:

1) *read*; 2) *write*; 3) *if*; 4) *then*; 5) *else*; 6) *for*; 7) *to*; 8) *while*; 9) *do*; 10) *true*; 11) *false*; 12) *or*; 13) *and*; 14) *not*; 15) *as*.

Таблица ограничителей содержит:

1) { ; 2) } ; 3) % ; 4) ! ; 5) $; 6) , ; 7) ; ; 8) [; 9)] ; 10) : ; 11) (; 12)) ; 13) +; 14) - ; 15) * ; 16) / ; 17) = ; 18) <> ; 19) > ; 20) < ; 21) <= ; 22) >= ; 23) /* ; 24) /* .

Таблицы идентификаторов и чисел формируются в ходе лексического анализа.

Пример 2.6. Описать результаты работы лексического анализатора для модельного языка *M*.

Входные данные ЛА: { % *k*, *sum*; *k as* 0;…

Выходные данные ЛА: (2,1) (2,3) (4,1) (2,6) (4,2) (2,7) (4,1) (1,15) (3,1) (2,7) …

Анализ текста проводится путем разбора по регулярным грамматикам и опирается на способ разбора по диаграмме состояний, снабженной дополнительными пометками-действиями. В диаграмме состояний с действиями каждая дуга имеет вид, представленный на рисунке 2.4. Смысл этой конструкции: если текущим является состояние *A* и очередной входной символ совпадает с t_i для какого либо *i*, то осуществляется переход в новое состояние *B*, при этом выполняются действия D_1, D_2, …, D_m.

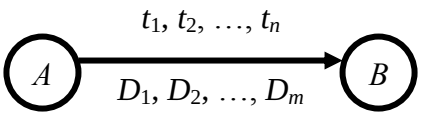


Рисунок 2.4 – Дуга ДС с действиями

Для удобства разбора вводится дополнительное состояние диаграммы *ER*, попадание в которое соответствует появлению ошибки в алгоритме разбора. Переход по дуге, не помеченной ни одним символом, осуществляется по любому другому символу, кроме тех, которыми помечены все другие дуги, выходящие из данного состояния.

Алгоритм 2.1. Разбор цепочек символов по ДС с действиями

Шаг 1. Объявляем текущим начальное состояние ДС *H*.

Шаг 2. До тех пор, пока не будет достигнуто состояние *ER* или конечное состояние ДС, считываем очередной символ анализируемой строки и переходим из текущего состояния ДС в другое по дуге, помеченной этим символом, выполняя при этом соответствующие действия. Состояние, в которое попадаем, становится текущим.

ЛА строится в два этапа:

1) построить ДС с действиями для распознавания и формирования внутреннего представления лексем;

2) по ДС с действиями написать программу сканирования текста исходной программы.

Пример 2.7. Составим ЛА для модельного языка *M*. Предварительно введем обозначения для переменных, процедур и функций.

Шаг 1. Построим ДС с действиями для распознавания и формирования внутреннего представления лексем модельного языка *M* (рисунок 2.5).

В диаграмме использованы следующие обозначения.

Переменные:

1) *CH* – очередной входной символ;

2) *S* - буфер для накапливания символов лексемы;

3) *B* – переменная для формирования числового значения константы;

4) *CS* - текущее состояние буфера накопления лексем с возможными значениями: *H* – начало; *I* – идентификатор; *N*2 – двоичное число; *N*8 – восьмеричное число; *N*10 – десятичное число; *N*16 – шестнадцатеричное число; *C*1, *C*2, *C*3 – комментарий;

*M*1 – меньше, меньше или равно, не равно; *M*2 – больше, больше или равно; *P*1 – точка; *P*2 – дробная часть числа; *B* – символ «*B*» или «*b*»; *O* – символ «*O*» или «*o*»; *D* – символ «*D*» или «*d*»; *HX* – символ «*H*» или «*h*»; *E*11 - символ «*E*» или «*e*»; *E*12, *E*13, *E*22 – порядок числа в экспоненциальной форме; *ZN*, *E*21 – знак порядка числа в экспоненциальной форме; *OG* – ограничитель; *V* – выход; *ER* –ошибка;

5) *t* - таблица лексем анализируемой программы с возможными значениями: *TW* - таблица служебных слов *M*-языка, *TL* – таблица ограничителей *M*-языка, *TI* - таблица идентификаторов программы, *TN* – таблица чисел, используемых в программе;

6) *z* - номер лексемы в таблице *t* (если лексемы в таблице нет, то *z*=0).

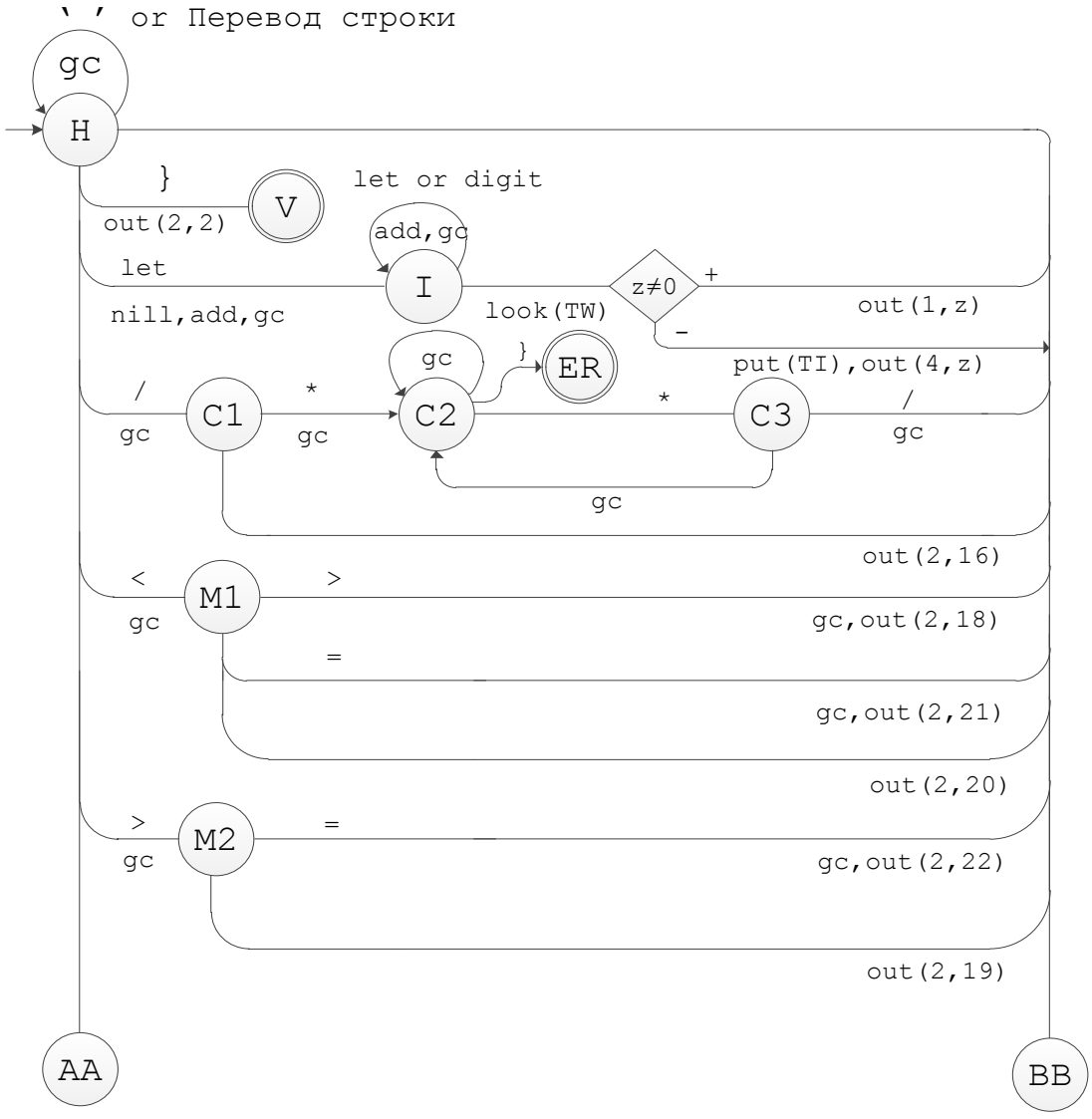


Рисунок 2.5 – Диаграмма состояний с действиями для модельного языка *M*

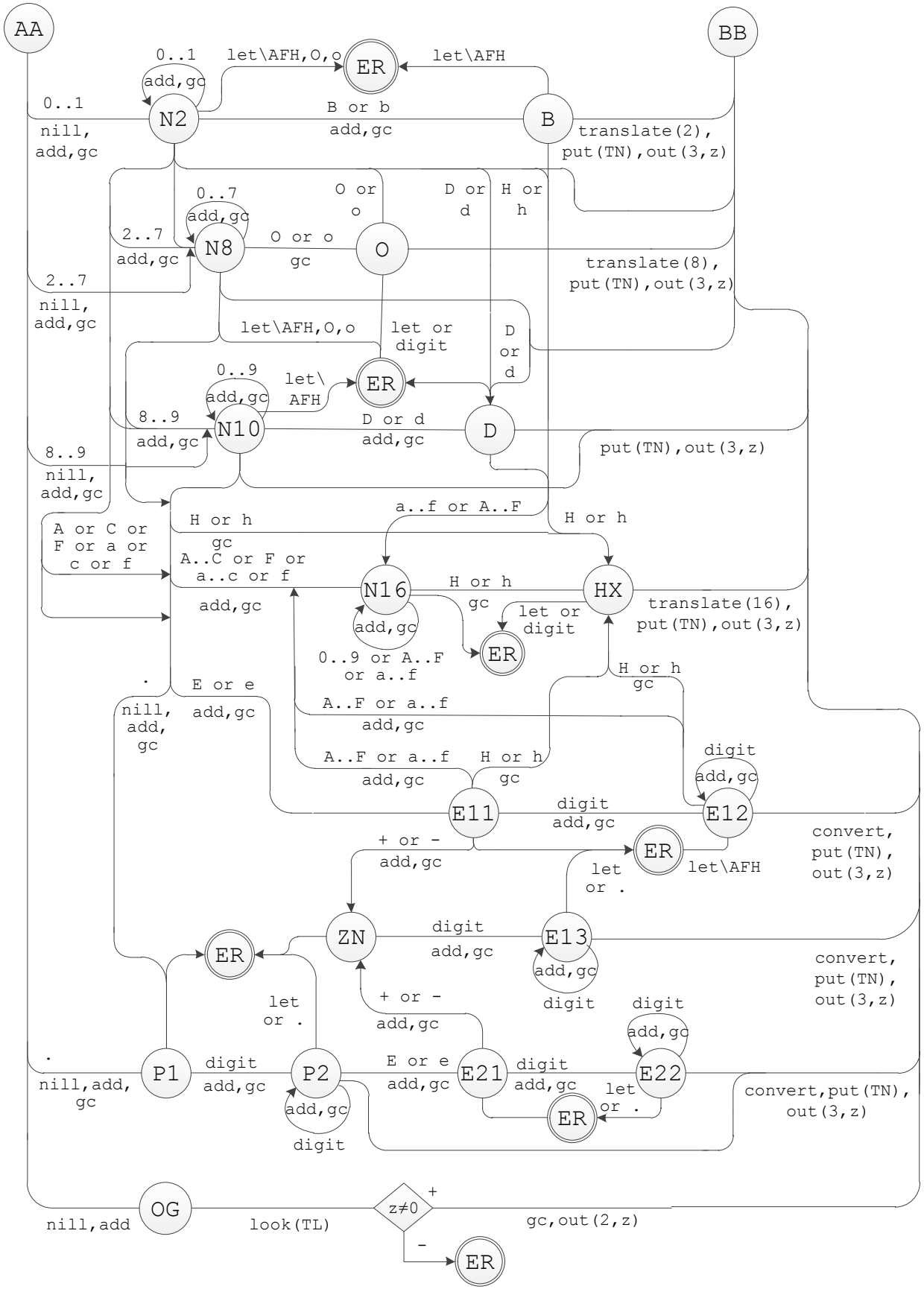


Рисунок 2.5 – Диаграмма состояний с действиями для модельного языка *M*, лист 2

Процедуры и функции:

1) *gc* – процедура считывания очередного символа текста в переменную *CH*;

2) *let* – логическая функция, проверяющая, является ли переменная *CH* буквой;

3) *digit* - логическая функция, проверяющая, является ли переменная *CH* цифрой;

4) *nill* – процедура очистки буфера *S*;

5) *add* – процедура добавления очередного символа в конец буфера *S*;

6) *look*(*t*) – функция поиска лексемы из буфера *S* в таблице *t* с возвращением номера лексемы в таблице;

7) *put*(*t*) – процедура записи лексемы из буфера *S* в таблицу *t*, если там не было этой лексемы; возвращает номер данной лексемы в таблице;

8) *out*(*n, k*) – процедура записи пары чисел (*n, k*) в файл лексем;

9) *check_hex* – логическая функция, проверяющая, является ли переменная *CH* цифрой или буквой из диапазона *A..F* или *a..f*;

10) *AFH* – логическая функция, проверяющая является ли переменная *CH* буквой из диапазона *A..F* и *a..f* или буквой *H*, *h*;

11*)* *translate*(*base*) – процедура перевода числа в буфере из системы счисления по основанию *base* в десятичную систему счисления;

12) *convert* – процедура преобразования действительного числа в буфере из строковой формы записи в десятичную форму.

Шаг 2. Составляем функцию *scanner* для анализа текста исходной программы:

```
bool scanner(){
   sost CS; gc(); CS=H;
   do{switch(CS){
        case H:{while(CH==' ' || CH=='\n' && !fcin.eof() )
                    gc();
               if(fcin.eof() )
                  CS=ER;
               if(let() ){nill(); add();
```

```
                gc(); CS=I;}
        else if(CH=='0' || CH=='1'){nill(); CS=N2; add(); gc();}
             else if(CH>='2' && CH<='7')
                  {nill(); CS=N8; add(); gc();}
                 else if(CH>='8' && CH<='9'){nill(); CS=N10; add(); gc();}
                      else if(CH=='.'){nill(); add(); gc(); CS=P1;}
                           else if(CH=='/'){gc(); CS=C1;}
                                else if(CH=='<'){gc(); CS=M1;}
                                     else if(CH=='>'){gc(); CS=M2;}
                                          else if(CH=='}'){out(2, 2); CS=V;}
                                               else CS=OG;
    break;}
case I:{while(let() || digit()){add(); gc();}
         look(TW);
         if(z!=0){out(1, z); CS=H;}
         else{put(TI); out(4, z); CS=H;}
        break;}
case N2:{while(CH=='0' || CH=='1'){ add(); gc();}
            if(CH>='2' && CH<='7')
            CS=N8;
            else if(CH=='8' || CH=='9')
              CS=N10;
            else if(CH=='A' || CH=='a' || CH=='C' || CH=='c' ||
                                          CH=='F' || CH=='f')
                 CS=N16;
               else if(CH=='E' || CH=='e'){add(); gc(); CS=E11;}
                    else if(CH=='D' || CH=='d'){add; gc(); CS=D;}
                         else if(CH=='O' || CH=='o')
                              CS=O;
                              else if(CH=='H' || CH=='h'){gc(); CS=HX;}
```

```
                    else if(CH=='.'){add(); gc(); CS=P1;}
                        else if(CH=='B' || CH=='b')
                            {add(); gc();CS=B;}
                        else if(let() )
                            CS=ER;
                            else CS=N10;
        break;}
case N8:{while(CH>='2' && CH<='7'){add(); gc();}
    if(CH=='8' || CH=='9')
        CS=N10;
    else if(CH=='A' || CH=='a' || CH=='B' || CH=='b' || CH=='C' ||
                            CH=='c' ||CH=='F' || CH=='f')
        CS=N16;
      else if(CH=='E' || CH=='e'){add(); gc(); CS=E11;}
        else if(CH=='D' || CH=='d'){add(); gc(); CS=D;}
          else if(CH=='H' || CH=='h'){gc(); CS=HX;}
            else if(CH=='.'){add(); gc(); CS=P1;}
              else if(CH=='O' || CH=='o'){gc();CS=O;}
                else if(let() )
                    CS=ER;
                  else CS=N10;
    break;}
case N10:{
    while(CH=='8' || CH=='9'){add(); gc();}
    if(CH=='A' || CH=='a' || CH=='B' || CH=='b' || CH=='C' ||
                        CH=='c' || CH=='F' || CH=='f')
        CS=N16;
    else if(CH=='E' || CH=='e'){add(); gc(); CS=E11;}
        else if(CH=='H' || CH=='h'){gc(); CS=HX;}
          else if(CH=='.'){add(); gc(); CS=P1;}
```

```
                    else if(CH=='D' || CH=='d'){add();gc();CS=D;}
                        else if(let() )
                            CS=ER;
                            else { put(TN); out(3,z); CS=H;}
        break;}
        case N16:{while(check_hex() ){add(); gc();}
        if(CH=='H' || CH=='h'){gc(); CS=HX;}
        else CS=ER;
        break;}
case B:{if( check_hex() )
           CS=N16;
        else if( CH=='H' || CH=='h'){gc(); CS=HX;}
            else if(let() )
                CS=ER;
               else{translate(2); put(TN); out(3,z); CS=H;}
        break;}
case O:{if( let() || digit())
          CS=ER;
      else {translate(8); put(TN);out(3,z);CS=H;}
      break;}
case D:{if(CH=='H' || CH=='h'){gc(); CS=HX;}
      else if(check_hex() )
             CS=N16;
          else if(let() )
              CS=ER;
             else{put(TN); out(3,z); CS=H;}
      break;}
case HX:{if(let() || digit() )
          CS=ER;
      else{translate(16); put(TN);out(3,z); CS=H;}
```

```
        break;}
    case E11:{if(digit() ){add(); gc(); CS=E12;}
        else if(CH=='+' || CH=='-'){add(); gc(); CS=ZN;}
            else if(CH=='H' || CH=='h'){gc(); CS=HX;}
                else if(check_hex() ){add(); gc(); CS=N16;}
                    else CS=ER;
        break;}
    case ZN:{if(digit() ){add(); gc(); CS=E13;}
        else CS=ER;
        break;}
    case E12:{while(digit() ){add(); gc();}
        if(check_hex() )
          CS=N16;
        else if(CH=='H' || CH=='h'){gc(); CS=HX;}
            else if(let() )
                 CS=ER;
                else{convert(); put(TN); out(3,z); CS=H;}
        break;}
    case E13:{while(digit() ){add(); gc();}
        if (let() || CH=='.')
           CS=ER;
        else {convert(); put(TN); out(3,z); CS=H;}
        break;}
    case P1:{if(digit() ) CS=P2; else CS=ER; break;}
    case P2:{while(digit() ){add(); gc();}
        if (CH=='E' || CH=='e'){add(); gc(); CS=E21;}
        else if (let() || CH=='.' )
             CS=ER;
            else {convert();put(TN); out(3,z); CS=H;}
        break;}
```

```
case E21:{if (CH=='+' || CH=='-') {add(); gc(); CS=ZN;}
    else if (digit() )
          CS=E22;
        else CS=ER;
    break;}
case E22:{while(digit() ){add(); gc();}
    if (let() || CH=='.')
       CS=ER;
    else {convert(); put(TN); out(3,z); CS=H;}
    break;}
case C1:{if (CH=='*'){gc(); CS=C2;}
    else{out(2,16); CS=H;}
    break;}
case C2:{int flag=0;
    while(CH!='*' && !flag && CH!='}'){flag=gc();}
    if (CH=='}' || flag) CS=ER;
     else {gc(); CS=C3;}
    break;}
case C3:{if (CH=='/'){gc(); CS=H;}
    else CS=C2;
    break;}
case M1:{if (CH=='>') {gc(); out(2, 18); CS=H;}
    else if (CH=='=') {gc(); out(2, 21); CS=H;}
        else {out(2, 20); CS=H;}
    break;}
case M2:{if (CH=='='){gc(); out(2, 22); CS=H;}
            else {out(2,19); CS=H;}
          break;}
case OG:{nill(); add();
          look(TL);
```

```
                    if(z!=0){gc();
                            out(2,z); CS=H;}
                      else CS=ER;
                    break;}
      } // end switch
    }while (CS!=V && CS!=ER);
  return CS;
      }
```

2.4 Синтаксический анализатор программы

Задача синтаксического анализатора (СиА) - провести разбор текста программы, сопоставив его с эталоном, данным в описании языка. Для синтаксического разбора используются контекстно-свободные грамматики (КС-грамматики).

Один из эффективных методов синтаксического анализа – метод рекурсивного спуска. В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).

Пример 2.8. Дана грамматика $G(\{a, b, c, \perp\}, \{S, A, B\}, P, S)$ с правилами P: 1) $S \to AB\perp$; 2) $A \to a$; 3) $A \to cA$; 4) $B \to bA$. Требуется выполнить анализ строки $cabca\perp$.

Левосторонний вывод цепочки имеет вид:

$S \Rightarrow AB\perp \Rightarrow cAB\perp \Rightarrow caB\perp \Rightarrow cabA\perp \Rightarrow cabcA\perp \Rightarrow cabca\perp$.

Нисходящее дерево разбора цепочки представлено на рисунке 2.6.

Метод рекурсивного спуска реализует разбор цепочки сверху вниз следующим образом. Для каждого нетерминального символа грамматики создается своя процедура, носящая его имя. Задача этой процедуры – начиная с указанного места

исходной цепочки, найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибок, которая выдает сообщение о том, что цепочка не принадлежит языку грамматики и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры составляется непосредственно по правилам вывода соответствующего нетерминала, при этом терминалы распознаются самой процедурой, а нетерминалам соответствуют вызовы процедур, носящих их имена.

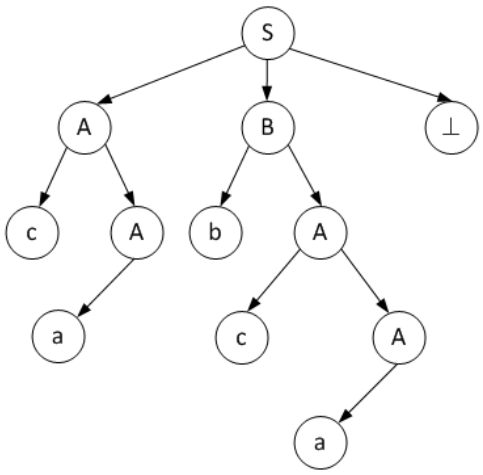


Рисунок 2.6 – Дерево нисходящего разбора цепочки *cabca*⊥

Пример 2.9. Построим синтаксический анализатор методом рекурсивного спуска для грамматики G из примера 2.8.

Введем следующие обозначения:

1) *CH* – текущий символ исходной строки;
2) *gc* – процедура считывания очередного символа исходной строки в переменную *CH*;
3) *Err* - процедура обработки ошибок, возвращающая по коду соответствующее сообщение об ошибке.

С учетом введенных обозначений, процедуры синтаксического разбора будут иметь вид.

```
void S
{A; B;
    if (CH != '}' ) ERR;
}
void A
{if (CH== 'a' ) gc;
    else if(CH=='c') {gc; A;}
else Err;
}
```

```
void B
{if (CH== 'b')
   {
    gc; B;
    }
    else Err;
}
```

Теорема 2.1. Достаточные условия применимости метода рекурсивного спуска

Метод рекурсивного спуска применим к грамматике, если правила вывода грамматики имеют один из следующих видов:

1) $A \to \alpha$, где $\alpha \in (T \cup N)^*$, и это единственное правило вывода для этого нетерминала;

2) $A \to a_1\alpha_1 \mid a_2\alpha_2 \mid \ldots \mid a_n\alpha_n$, где $a_i \in T$ для каждого $i=1, 2,\ldots, n$; $a_i \neq a_j$ для $i \neq j$, $\alpha_i \in (T \cup N)^*$, т.е. если для нетерминала A несколько правил вывода, то они должны начинаться с терминалов, причем эти терминалы должны быть различными.

Данные требования являются достаточными, но не являются необходимыми. Можно применить эквивалентные преобразования КС-грамматик, которые способствуют приведению грамматики к требуемому виду, но не гарантируют его достижения.

При описании синтаксиса языков программирования часто встречаются правила, которые задают последовательность однотипных конструкций, отделенных друг от друга каким-либо разделителем. Общий вид таких правил:

$L \to a \mid a, L$ или в РБНФ $\langle L \rangle ::= a \{, a\}$.

Формально здесь не выполняются условия метода рекурсивного спуска, т.к. две альтернативы начинаются одинаковыми терминальными символами. Но если принять соглашения, что в подобных ситуациях выбирается самая длинная подцепочка, выводимая из нетерминала *L*, то разбор становится детерминированным, и метод рекурсивного спуска будет применим к данному правилу грамматики. Соответствующая правилу процедура будет иметь вид:

```
void L
{ if (CH != ‘a’) Err;
    else gc;
    while (CH = =’,’ )
       { gc;
          if (CH != ’a’ ) Err;
           else gc;
       }
}
```

Пример 2.10. Построим синтаксический анализатор методом рекурсивного спуска для модельного языка *M*.

Вход – файл лексем в числовом представлении.

Выход – заключение о синтаксической правильности программы или сообщение об имеющихся ошибках.

Введем обозначения:

1) *LEX* – переменная, содержащая текущую лексему, считанную из файла лексем;

2) *gl* – процедура считывания очередной лексемы из файла лексем в переменную *LEX*;

3) EQ(*S*) – логическая функция, проверяющая, является ли текущая лексема *LEX* лексемой для *S*;

4) ID – логическая функция, проверяющая, является ли *LEX* идентификатором*;*

5) *NUM* - логическая функция, проверяющая, является ли *LEX* числом.

6) *err_proc*(*numb_error*) – процедура, обрабатывающая ошибку с номером *numb_error*.

Процедуры проверки выполнения правил, описывающих язык *M* и составляющие синтаксический анализатор, будут иметь следующий вид:

1) для правила $PR \rightarrow \{BODY\}$

```
void PR()
{ gl();
  if (EQ("{")) {gl(); BODY();
                  if (EQ("}") = = 0) err_proc(3);}
   else err_proc(2);
}
```

2) для правила $BODY \rightarrow (DESCR \mid OPER)\ \{;\ (DESCR \mid OPER)\}$

```
void BODY()
{ if (EQ("%") || EQ("#") || EQ("$") ) DESCR ();
 else if(ID() || EQ("if") || EQ("[") || EQ("while") || EQ("for"
        || EQ("read") || EQ("write") ) OPER();
     else err_proc(31);
     while(EQ(";") ) {gl;
                  if (EQ("%") || EQ("#") || EQ("$") ) DESCR ();
                  else if (ID() || EQ("if") || EQ("[") || EQ("while")
                          || EQ("for") || EQ("read") || EQ("write") ) OPER();
                      else err_proc(31);
                   }
}
```

3) для правила $DESCR \rightarrow (\% \mid \# \mid \$)\ ID_1$

```
void DESCR()
{ if (EQ("%") ) {gl(); ID1();}
 else if (EQ("#") ) {gl(); ID1();}
     else if (EQ("$") ) {gl(); ID1();}
}
```

4) для правила $ID_1 \rightarrow ID$ {, *ID*}

```
void ID1()
{if (ID() ){gl();
        while(EQ(",") ) {gl(); if (ID() ) gl();
                                else err_proc(18);
                        }
        }
    else err_proc(19);
}
```

5) для правила *FACT*→ *ID* | *NUM* | *LOG* | *not FACT* | (*COMPARE*)

```
void FACT()
{if (ID() || NUM() || EQ("true") || EQ("false") ) gl();
 else if (EQ("not") ){gl(); FACT();}
     else if (EQ("(") ){gl(); COMPARE();
                        if(EQ(")") ) gl();
                        else err_proc(16);
                        }
          else err_proc(17);
}
```

Аналогично составляются оставшиеся процедуры.

2.5 Семантический анализатор программы

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой. Эти правила носят контекстно-зависимый характер, их называют семантическими соглашениями или контекстными условиями.

Рассмотрим пример построения семантического анализатора (СеА) для программы на модельном языке *M*. Соблюдение контекстных условий для языка *M* предполагает три типа проверок:

1) обработка описаний;

2) анализ выражений;

3) проверка правильности операторов.

В оптимизированном варианте СиА и СеА совмещены и осуществляются параллельно. Поэтому процедуры СеА будем внедрять в ранее разработанные процедуры СиА.

Вход: файл лексем в числовом представлении.

Выход: заключение о семантической правильности программы или о типе обнаруженной семантической ошибки.

Обработка описаний

Задача обработки описаний - проверить, все ли переменные программы описаны правильно и только один раз. Эта задача решается следующим образом.

Таблица идентификаторов, введенная на этапе лексического анализа, расширяется, приобретая вид таблицы 2.1. Описание таблицы идентификаторов будет иметь вид:

```
Struct Tabid
    {char id[1024], typid[4];
       bool descrid, nb;
       double nd;
       int ni, addrid;
    };
   typedef struct Tabid tabid;
   tabid mTI[s_stack];
```

Таблица 2.1 – Таблица идентификаторов на этапе СеА

Номер	Идентификатор	Описан	Тип	Адрес
1	*K*	1	%	…
2	*Sum*	0	…	…

Поле «описан» таблицы на этапе лексического анализа заполняется нулем, а при правильном описании переменных на этапе семантического анализа заменяется единицей.

При выполнении процедуры *ID*1 вводится стековая переменная-массив, в которую заносится контрольное число 0. По мере успешного выполнения процедуры *ID*1 в стек заносятся номера считываемых из файла лексем, под которыми они записаны в таблице идентификаторов. Как только при считывании, следующая лексема не «,» из стека извлекаются записанные номера и по ним в таблице идентификаторов проставляется 1 в поле «описан» (к этому моменту там должен быть 0). Если очередная лексема будет «%», «#» или «$», то попутно в таблице идентификаторов поле «тип» заполняется соответствующим типом.

Пример 2.11. Пусть фрагмент описания на модельном языке имеет вид: % *k, sum* … Тогда соответствующий фрагмент файла лексем: (2, 3) (4, 1) (2, 6) (4, 2)…Содержимое стека при выполнении процедуры *DESCR* представлено на рисунке 2.7.

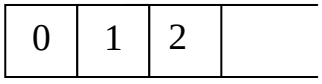


Рисунок 2.7 – Содержимое стека при выполнении процедуры *DESCR*

Для реализации обработки описаний введем следующие обозначения переменных и процедур:

1) *LEX* – переменная, хранящая значение очередной лексемы, представляющая собой структуру с 2 полями, т.е. для лексемы (*n*, *k*) *LEX.n_tabl=n*, *LEX.leksema=k*;

2) *gl* – процедура считывания очередной лексемы в переменную *LEX*;

3) *inst*(*l*) - процедура записи в стек числа *l*;

4) *outst*(*l*) – процедура вывод из стека числа *l*;

5) *instl* – процедура записи в стек номера, под которым лексема хранится в таблице идентификаторов, т.е. *inst*(*LEX*);

6) *dec*(*t*) - процедура вывода всех чисел из стека и вызова процедуры *decid*(*l*, *t*);

7) *decid*(*l*, *t*) – процедура проверки и заполнения поля «описан» и «тип» таблицы идентификаторов для лексемы с номером *l* и типа *t.*

Процедуры *dec* и *decid* имеют вид:

```
void decid (… l, ... t)
{ if(mTI[l].descrid==1) err_proc(20);
   else {mTI[l].descrid=1; strcpy(mTI[l].typid, t);}
 }
void dec(... t)
{ char l[s_stack];
  outst(l);
 while l[0] != 0
        { strcpy(stack, l);
        decid(look(4)-1, t);
            outst(l);
        }
}
```

Правила и процедуры *DESCR и ID*1 с учетом семантических проверок принимают вид:

$DESCR \rightarrow$ (% <*dec*(“%”)> | # <*dec*(“#”)> | $ <*dec*(“$”)>) ID_1

```
void D
{ if(EQ("%") ){gl(); ID1(); dec("%");}
 else if(EQ("#") ){gl(); ID1(); dec("#");}
     else if(EQ("$") ){gl(); ID1(); dec("$");}
}
```

$ID_1 \rightarrow$ <*inst*(0)> *I* <*instl*> {, *I* <*instl*> }

```
void ID1()
{ inst(0);
    if (ID() ){instl(); gl();
                while(EQ(",") ){gl(); if(ID() ){instl(); gl();}
                                    else err_proc(18); }
              }
    else err_proc(19);}
```

Анализ выражений

Задача анализа выражений - проверить описаны ли переменные, встречающиеся в выражениях, и соответствуют ли типы операндов друг другу и типу операции.

Эти задачи решаются следующим образом. Вводится таблица двуместных операций (таблица 2.2) и стек, в который в соответствии с разбором выражения *E* заносятся типы операндов и знак операции. После семантической проверки в стеке остается только тип результата операции. В результате разбора всего выражения в стеке остается тип этого выражения.

Таблица 2.2 – Фрагмент таблицы двуместных операций *TOP*

Операция	Тип 1	Тип 2	Тип результата
+	%	%	%
>	%	%	$
…	…	...	...

Для реализации анализа выражений введем следующие обозначения процедур и функций:

1) *checkid* - процедура, которая для лексемы *LEX*, являющейся идентификатором, проверяет по таблице идентификаторов *TI*, описан ли он, и, если описан, то помещает его тип в стек;

2) *checkop* – процедура, выводящая из стека типы операндов и знак операции, вызывающая процедуру *gettype*(*op*, *t*1, *t*2, *t*), проверяющая соответствие типов и записывающая в стек тип результата операции;

3) *gettype*(*op*, *t*1, *t*2, *t*) – функция, которая по таблице операций *TOP* для операции *op* выдает тип *t* результата и типы *t*1, *t*2 операндов. Возвращает *false*, если операция не найдена;

4) *checknot* – процедура проверки типа для одноместной операции «*not*».

5) *check_type*(*type*) – логическая функция, проверяющая соответствие типа в верхушке стека типу *type*.

Перечисленные процедуры имеют следующий вид:

void checkid()

```
{char k[s_stack];
      strcpy(k, tbl[lex.n_tabl][lex.leksema-1].c_str() );
      if (mTI[lex.leksema-1].descrid==0) err_proc(21);
      inst(mTI[lex.leksema-1].typid);
}
void checkop()
{outst(top2); outst(op); outst(top1);
 strcpy(t1, top1); strcpy(t2, top2);
 if (gettype(op, t1, t2, t) = = 0) err_proc(22);
 inst(t);
}
void checknot()
{char t[s_stack];
 outst(t);
 if (strcmp(t,"$")!=0 ) err_proc(23);
 inst(t);
}
bool check_type(type)
{ if (strcmp(stack,"%") = = 0) return true;
  i=look(4);
 if (i>0) return strcmp(type, mTI[i].typid) = = 0;
 return 0;
}
```

Правила, описывающие выражения языка *M*, расширенные процедурами семантического анализа, принимают вид.

COMPARE → *ADD* {(> | >= | <= | < | <> | =) <*instl*> *COMPARE* <*checkop*>}

ADD → *MULT* {(+ | - | *or*) <*instl*> *ADD* <*checkop*>}

MULT → *FACT* {(* | / | *and*) <*instl*> *MULT* <*checkop*>}

FACT → *ID* <*checkid*> | *NUM* <*inst*("%")> | *NUM* <*inst*("#")> |
NUM <*inst*("$")> | *LOG* <*inst*('$')>| *not FACT* <*checknot*>| (*COMPARE*)

Пример 2.12. Дано выражение *a*+5**b*. Дерево разбора выражения и динамика содержимого стека представлены на рисунке 2.8.

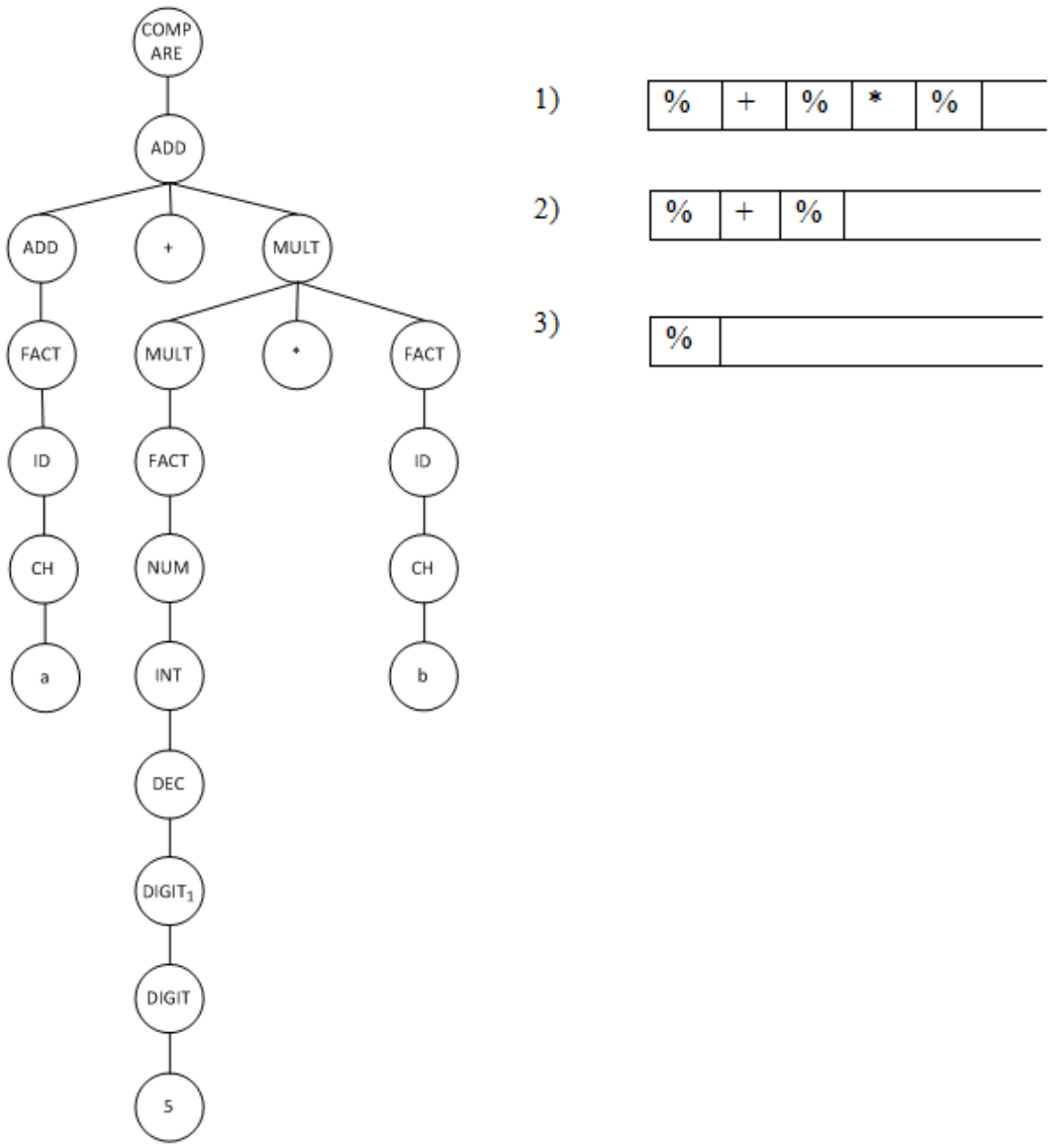


Рисунок 2.8 – Анализ выражения *a*+5**b*

Данные задачи решаются путем включения в правило *OPER* ранее рассмотренной процедуры *checkid*, а также новых процедур *eqtype* и *eqbool*, имеющих следующий вид:

```
void eqtype()
{outst(t2); outst(t1);
    if (strcmp(t1,"#") = = 0 && strcmp(t2,"%") = = 0) 1;
    else if (strcmp(t1, t2)!=0) err_proc(24); }
void eqbool()
```

{*outst*(*t*);

if(*strcmp*(*t*,"$")!=0) *err_proc*(25); }

Правило *OPER* с учетом процедур СеА примет вид:

OPER → [$OPER_1$] | *ID* <*checkid*> *as COMPARE* <*eqtype*> |

if COMPARE <*eqbool*> *then OPER* |

if COMPARE <*eqbool*> *then OPER else OPER* |

while COMPARE <*egbool*> *do OPER* |

for ID <*checkid*> <*checktype*(“%”)> *as COMPARE* <*eqtype*> *to COMPARE* <*check_type*(“%”)> *do OPER* |

write (*EXPR*) | *read* (ID_1)

3 Постановка задачи к курсовой работе

Разработать распознаватель модельного языка программирования, выполнив следующие действия.

1) В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

а) РБНФ;

б) диаграмм Вирта;

в) формальных грамматик.

2) Написать пять содержательных примеров программ, раскрывающих особенности конструкций модельного языка программирования, отразив в этих примерах все его функциональные возможности.

3) Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.

4) По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.

5) Разработать программное средство, реализующее лексический анализ текста программы на входном языке.

6) Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.

7) Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.

8) Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.

9) Распечатать пример таблиц идентификаторов и двуместных операций.

10) Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.

11) Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.

4 Требования к содержанию курсовой работы

Курсовая работа должна иметь следующую структуру и состоять из разделов.

Введение

1 Постановка задачи

2 Формальная модель задачи

3 Спецификация основных процедур и функций

3.1 Лексический анализатор

3.2 Синтаксический анализатор

3.3 Семантический анализатор

4 Структурная организация данных

4.1 Спецификация входных данных

4.2 Спецификация выходных данных

5 Разработка алгоритма решения задачи

5.1 Укрупненная схема алгоритма программного средства

5.2 Детальная разработка алгоритмов отдельных подзадач

6 Установка и эксплуатация программного средства

7 Работа с программным средством

Заключение

Список использованных источников

Приложение А – Текст программы

Приложение Б – Контрольный пример

Введение. Во введении кратко описывается состояние вопроса разработки распознавателей формальных языков, формулируются цель и задачи курсовой работы, а также актуальность и обоснованность их решения.

Постановка задачи. Поставленная преподавателем задача разбивается на ряд подзадач, которые необходимо решить для достижения цели курсовой работы.

Формальная модель задачи. Данный раздел содержит положения из теории формальных языков, грамматик и автоматов, лежащие в основе разработки распознавателя модельного языка.

Спецификации основных процедур и функций. Для каждой программной единицы необходимо представить входные данные, функции, которые выполняются, и результаты ее работы.

Разработка алгоритма решения задачи. На основе анализа всех функций, которые должно выполнять проектируемое программное средство, необходимо разработать и описать алгоритм решения задачи. В зависимости от выполнения или невыполнения тех или иных условий, показать порядок и последовательность решения задачи. Логическую структуру программного средства представить с помощью укрупненной схемы алгоритма.

Детальная разработка алгоритмов отдельных подзадач. В этом разделе должна быть представлена логическая структура модулей и процедур, составляющих данное программное средство. Для модулей, которые имеют сложную логическую структуру, описание может быть иллюстрировано схемой алгоритма.

Структурная организация данных. В этом разделе необходимо описать данные, используемые в программном средстве (файлы, массивы и т.д.) их структу-

ру, типы и т.д. Если данные имеют сложную структуру, то описание необходимо пояснять графическими схемами.

Установка программного средства. Описываются все действия, необходимые для установки программного средства (ПС) на ЭВМ. Также объем, занимаемый ПС на жестком магнитном диске, минимальный объем оперативной памяти, необходимый для его эксплуатации, и другие технические характеристики оборудования.

Работа с программным средством. Здесь поясняется обращение к программе, способы передачи управления, вызов программы и др. Должна быть описана последовательность выполнения работы, средства защиты, разработанные в данном ПС, реакция ПС на неверные действия пользователя.

Заключение. В заключении приводятся основные выводы и перспективы дальнейшего развития представленного ПС.

Список использованных источников представляет собой перечень всей литературы, которая была использована при разработке ПС и оформлении документации на него. Список использованных источников формируется в том порядке, в котором были ссылки на использованную литературу, с указанием издательства, года издания и количества листов в книге согласно стандарту предприятия.

Приложения должны содержать текст ПС, контрольные и тестовые примеры, результаты работы ПС.

5 Индивидуальные варианты задания

Операции языка (первая цифра варианта) представлены в таблицах 5.1 – 5.4.

Таблица 5.1 - Операции группы «отношение»

Номер	Синтаксис группы операций (в порядке следования: неравно, равно, меньше, меньше или равно, больше, больше или равно)
1	<операции_группы_отношения>:: = < > | = | < | <= | > | >=
2	<операции_группы_отношения>:: = != | = = | < | <= | > | >=
3	<операции_группы_отношения>::= *NE* | *EQ* | *LT* | *LE* | *GT* | *GE*

Таблица 5.2 - Операции группы «сложение»

Номер	Синтаксис группы операций (в порядке следования: сложение, вычитание, дизъюнкция)
1	<операции_группы_сложения>:: = + | - | *or*
2	<операции_группы_сложения>:: = + | - | ||
3	<операции_группы_сложения>:: = *plus* | *min* | *or*

Таблица 5.3 - Операции группы «умножение»

Номер	Синтаксис группы операций (в порядке следования: умножение, деление, конъюнкция)
1	<операции_группы_умножения>::= * | / | *and*
2	<операции_группы_умножения>:: = * | / | &&
3	<операции_группы_умножения>::= *mult* | *div* | *and*

Таблица 5.4 - Унарная операция

Номер	Синтаксис операции
1	<унарная_операция>::= *not*
2	<унарная_операция>::= !
3	<унарная_операция>::= ~

Выражения языка задаются правилами:

<выражение>::= <операнд>{<операции_группы_отношения> <операнд>}

<операнд>::= <слагаемое> {<операции_группы_сложения> <слагаемое>}

<слагаемое>::= <множитель> {<операции_группы_умножения> <множитель>}

<множитель>::= <идентификатор> | <число> | <логическая_константа> |

<унарная_операция> <множитель> | «(»<выражение>«)»

<число>::= <целое> | <действительное>

<логическая_константа>::= *true* | *false*

Правила, определяющие идентификатор, букву и цифру:

<идентификатор>::= <буква> {<буква> | <цифра>}

<буква>::= *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* |
U | *V* | *W* | *X* | *Y* | *Z* | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p*
q | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z*

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Правила, определяющие целые числа:

<целое>::= <двоичное> | <восьмеричное> | <десятичное> |
<шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (*B* | *b*)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (*O* | *o*)

<десятичное>::= {/ <цифра> /} [*D* | *d*]

<шестнадцатеричное>::= <цифра> {<цифра> | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* |
c | *d* | *e* | *f*} (*H* | *h*)

Правила, описывающие действительные числа:

<действительное>::= <числовая_строка> <порядок> |
[<числовая_строка>] . <числовая_строка> [порядок]

<числовая_строка>::= {/ <цифра> /}

<порядок>::= (E | e)[+ | -] <числовая_строка>

Правила, определяющие структуру программы (вторая цифра варианта), представлены в таблице 5.5.

Таблица 5.5 – Структура программы

Номер	Структура программы
1	<программа>::= *program var* <описание> *begin* <оператор> {; <оператор>} *end.*
2	<программа>::= «{» {/ (<описание> | <оператор>) ; /} «}»
3	<программа> = {/ (<описание> | <оператор>) (: | переход строки) /} *end*

Правила, определяющие раздел описания переменных (третья цифра варианта), показаны в таблице 5.6.

Таблица 5.6 - Синтаксис команд описания данных

Номер	Синтаксис команд описания данных
1	<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}
2	<описание>::= *dim* <идентификатор> {, <идентификатор> } <тип>
3	<описание>::= <тип> <идентификатор> { , <идентификатор> }

Правила, определяющие типы данных (четвертая цифра варианта), представлены в таблице 5.7.

Таблица 5.7- Описание типов данных

Номер	Описание типов (в порядке следования: целый, действительный, логический)
1	<тип>::= % | ! | $
2	<тип>::= *integer* | *real* | *boolean*
3	<тип>::= *int* | *float* | *bool*

Правило, определяющее оператор программы (пятая цифра варианта).

<оператор>::= <составной> | <присваивания> | <условный> |
<фиксированного_цикла> | <условного_цикла> | <ввода> |
<вывода>

Составной оператор описан в таблице 5.8.

Таблица 5.8 - Синтаксис составного оператора

Номер	Синтаксис оператора
1	<составной>::= «[» <оператор> { (: | перевод строки) <оператор> } «]»
2	<составной>::= *begin* <оператор> { ; <оператор> } *end*
3	<составной>::= «{» <оператор> { ; <оператор> } «}»

Оператор присваивания описан в таблице 5.9.

Таблица 5.9 - Синтаксис оператора присваивания

Номер	Оператор присваивания
1	<присваивания>::= <идентификатор> *as* <выражение>
2	<присваивания>::= <идентификатор> := <выражение>
3	<присваивания> ::= [*let*] <идентификатор> = <выражение>

Оператор условного перехода задан в таблице 5.10.

Таблица 5.10 - Синтаксис оператора условного перехода

Номер	Оператор условного перехода
1	<условный>::= *if* <выражение> *then* <оператор> [*else* <оператор>]
2	<условный>::= *if* «(»<выражение> «)» <оператор> [*else* <оператор>]
3	<условный>::= *if* <выражение> *then* <оператор> [*else* <оператор>] *end_else*

Оператор цикла с фиксированным числом повторений описан в таблице 5.11.

Таблица 5.11 - Синтаксис оператора цикла с фиксированным числом повторений

Номер	Синтаксис оператора
1	<фиксированного_цикла>::= *for* <присваивания> *to* <выражение> *do* <оператор>
2	<фиксированного_цикла>::= *for* <присваивания> *to* <выражение> [*step* <выражение>] <оператор> *next*
3	<фиксированного_цикла>::= *for* «(»[<выражение>] ; [<выражение>] ; [<выражение>] «)» <оператор>

Условный оператор цикла задан в таблице 5.12.

Таблица 5.12 - Синтаксис условного оператора цикла

Номер	Синтаксис оператора
1	<условного_цикла>::= *while* <выражение> *do* <оператор>
2	<условного_цикла>::= *while* «(»<выражение> «)» <оператор>
3	<условного_цикла>::= *do while* <выражение> <оператор> *loop*

Оператор ввода описан в таблице 5.13.

Таблица 5.13 - Синтаксис оператора ввода

Номер	Синтаксис оператора
1	<ввода>::= *read* «(»<идентификатор> {, <идентификатор> } «)»
2	<ввода>::= *readln* идентификатор {, <идентификатор> }
3	<ввода>::= *input* «(»<идентификатор> {пробел <идентификатор>} «)»

Оператор вывода представлен в таблице 5.14.

Таблица 5.14 - Синтаксис оператора вывода

Номер	Синтаксис оператора
1	<вывода>::= *write* «(»<выражение> {, <выражение> } «)»
2	<вывода>::= *writeln* <выражение> {, <выражение> }
3	<вывода>::= *output* «(»<выражение> { пробел <выражение> } «)»

Многострочные комментарии в программе (шестая цифра варианта) определены в таблице 5.15. Индивидуальные номера вариантов представлены в таблице 5.16.

Таблица 5.15 – Синтаксис многострочных комментариев

Номер	Признак начала комментария	Признак конца комментария
1	{	}
2	/*	*/
3	(*	*)

Таблица 5.16 – Индивидуальные номера вариантов

Номер варианта	Номер задания	Номер варианта	Номер здания
1	111111	16	223122
2	122211	17	223322

Продолжение таблицы 5.16

Номер варианта	Номер задания	Номер варианта	Номер здания
3	113211	18	231123
4	113311	19	232223
5	121132	20	233323
6	121212	21	311111
7	123112	22	311211
8	123312	23	311311
9	131111	24	332211
10	132111	25	313311
11	211121	26	321122
12	213222	27	321222
13	213321	28	323122
14	221122	29	331133
15	222222	30	331233

6 Контрольные вопросы для самопроверки

1) Назовите основные способы описания синтаксиса языков программирования.

2) Дайте определение понятия «формальная грамматика».

3) Перечислите основные метасимволы, используемые в РБНФ.

4) Какие графические примитивы используются в метаязыке диаграмм Вирта?

5) Изобразите схематично распознаватель языков и поясните принцип его функционирования.

6) Какие классы распознавателей языков выделяют?

7) Что называется лексемой языка программирования?

8) Какие задачи выполняет лексический анализатор программы?

9) Какой тип грамматик по классификации Хомского лежит в основе лексического анализа программы?

10) Перечислите основные группы лексем языков программирования.

11) Что представляет собой диаграмма состояний с действиями?

12) Расскажите алгоритм разбора цепочек по ДС с действиями.

13) Составьте диаграмму состояний с действиями для модельного языка.

14) Напишите функцию сканирования текста программы на модельном языке по ДС с действиями.

15) Каково назначение синтаксического анализатора программы?

16) Какой тип грамматик по классификации Хомского лежит в основе синтаксического анализа программы?

17) В чем сущность метода рекурсивного спуска?

18) Назовите необходимые условия применимости метода рекурсивного спуска.

19) Расскажите алгоритм построения дерева нисходящего разбора для цепочек грамматики.

20) Какой вывод цепочки грамматики называется левосторонним?

21) Перечислите основные задачи семантического анализатора.

22) Предложите один из возможных способов обработки описаний программы.

23) Запишите синтаксические правила модельного языка, дополненные процедурами семантического анализа программы.

7 Тесты для самопроверки

1 Язык, состоящий из пустой строки и всевозможных строк, содержащих строку нулей и последующую строку единиц той же длины:

а) $L=\{(01)^n \mid n>0\}$;

б) $L=\{(01)^n \mid n\geq 0\}$;

в) $L=\{0^n1^n \mid n\geq 0\}$;

г) $L=\{0^n1^n \mid n>0\}$;

д) $L=\{0^n1^m \mid n, m\geq 0\}$.

2 Строка вида *ababab* принадлежит языку:

а) $L=\{ab^n \mid n>0\}$;

б) $L=\{(ab)^n \mid n\geq 0\}$;

в) $L=\{a^nb^n \mid n\geq 0\}$;

г) $L=\{a^nb^n \mid n>0\}$;

д) $L=\{ab^n \mid n\geq 0\}$.

3 Порядок следования понятий от общего к частному:

а) сентенция в грамматике; цепочка алфавита; строка языка грамматики;

б) строка языка грамматики; цепочка алфавита; сентенция в грамматике;

в) цепочка алфавита; строка языка грамматики; сентенция в грамматике;

г) сентенция в грамматике; строка языка грамматики; цепочка алфавита;

д) цепочка алфавита; сентенция в грамматике; строка языка грамматики.

4 Множество правил вывода грамматики $G=(V_T, V_N, P, S)$ является конечным подмножеством множества:

а) $(V_T \cup V_N) \times (V_T \cup V_N)$;

б) $(V_T \cup V_N)^+ \times (V_T \cup V_N)^*$;

в) $(V_T \cup V_N)^* \times (V_T \cup V_N)^+$;

г) $V_N^* \times V_T^+$;

д) $V_T^+ \times V_N^*$.

5 Порядок следования значений конструкций БНФ от общему к частному:

а) {0 | 1}; {/0 | 1/}; (0 | 1);

б) {/0 | 1/}; {0 | 1}; (0 | 1);

в) (0 | 1); {0 | 1}; {/0 | 1/};

г) {0 | 1}; (0 | 1); {/0 | 1/};

д) {/0 | 1/}; (0 | 1); {0 | 1}.

6 Синтаксическая конструкция, которая может отсутствовать, в БНФ заключается в:

а) ( );

б) { };

в) « »;

г) < >;

д) [ ].

7 Ограничители вида {/ /} задают в расширенных БНФ:

а) итерацию ноль и более раз;

б) итерацию один и более раз;

в) ограничение нетерминала;

г) операцию альтернатива;

д) ограничение терминала.

8 Множеству строк, задаваемых конструкцией БНФ вида {/01/}, принадлежит строка:

а) ε;

б) 000111;

в) 010101;

г) 001100;

д) 001110.

9 В БНФ в угловые скобки < > заключаются:

а) терминалы;

б) нетерминалы;

в) альтернативы;

г) итерации;

д) необязательные конструкции.

10 Нетерминальным символам на диаграммах Вирта соответствуют:

а) кружки;

б) скругленные прямоугольники;

в) ветвления дуг;

г) прямоугольники;

д) слияния дуг.

11 Ветвления дуг на диаграммах Вирта задают:

а) нетерминальные символы;

б) терминальные символы;

в) постоянную группу терминальных символов;

г) итерации;

д) альтернативы.

12 Если для каждой конфигурации распознавателя существует не более одного возможного следующего шага, то управляющее устройство называется:

а) односторонним;

б) ограниченным;

в) детерминированным;

г) недетерминированным;

д) неограниченным.

13 Распознавателями для КС-языков являются:

а) МП-автоматы;

б) конечные автоматы;

в) машины Тьюринга;

г) линейные ограниченные автоматы;

д) линейные неограниченные автоматы.

14 Конечные автоматы являются распознавателями:

а) фразовых языков;

б) регулярных языков;

в) КС-языков;

г) КЗ-языков;

д) естественных языков.

15 Автомат $M(Q, T, F, H, Z)$ допускает строку τ, если существует путь по конфигурациям:

а) $(H, \tau) \vdash^* (q, \varepsilon)$ для некоторого $q \in Z$;

б) $(H, \tau) \vdash (q, \varepsilon)$ для некоторого $q \in Z$;

в) $(q, \tau) \vdash^* (q, \varepsilon)$ для некоторого $q \in Z$;

г) $(H, \varepsilon) \vdash^* (q, \tau)$ для некоторого $q \in Z$;

д) $(H, \varepsilon) \vdash (q, \tau)$ для некоторого $q \in Z$.

16 Язык L, принимаемый конечным автоматом $M(Q, T, F, H, Z)$ формально определяется как множество:

а) $L(M) = \{\tau \mid \tau \in Q^*$ и $(H, \tau) \vdash (q, \varepsilon)$ для некоторого $q \in Z\}$;

б) $L(M) = \{\tau \mid \tau \in T^*$ и $(H, \tau) \vdash^* (q, \varepsilon)$ для некоторого $q \in Z\}$;

в) $L(M) = \{\tau \mid \tau \in T^+$ и $(q, \tau) \vdash^* (q, \varepsilon)$ для некоторого $q \in Z\}$;

г) $L(M) = \{\tau \mid \tau \in T^*$ и $(q, \tau) \vdash^* (H, \varepsilon)$ для некоторого $q \in Z\}$;

д) $L(M) = \{\tau \mid \tau \in T^*$ и $(H, \varepsilon) \vdash^* (q, \tau)$ для некоторого $q \in Z\}$.

17 В грамматике $G=(\{a, b, c, d\}, \{A, C\}, \{A \to aACd \mid b;\ C \to c \mid \varepsilon\}, A)$ левосторонним является вывод:

а) $A \Rightarrow aACd \Rightarrow abCd \Rightarrow abdd$;

б) $A \Rightarrow aACd \Rightarrow aAd \Rightarrow abd$;

в) $A \Rightarrow aACd \Rightarrow aAcd \Rightarrow abcd$;

г) $A \Rightarrow aACd \Rightarrow abCd \Rightarrow abcd$;

д) $A \Rightarrow aACd \Rightarrow aAcd \Rightarrow acd$.

18 Процедура *procedure B*; *begin if CH= 'b' then begin gc*; *B end else Err end*; реализует рекурсивный спуск для:

а) правила $B \to bB \mid b$;

б) правила $B \to bB$;

в) правила $B \to b$;

г) правила $B \to Bb$;

д) правила $B \to Bb \mid b$.

19 Достаточные условия применимости метода рекурсивного спуска выполняются в правиле:

а) $S \to aS \mid aB$;

б) $S \to Sa \mid aB$;

в) $S \to Sa \mid a$;

г) $S \to aS \mid bB$;

д) $S \to aS \mid a$.

20 Этап разбора, на котором символы исходной программы, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку:

а) лексический анализ;

б) синтаксический анализ;

в) семантический анализ;

г) генерация кода;

д) оптимизация кода.

21 Лексический анализ проводится путем разбора по:

а) МП-автомату;

б) конечному автомату;

в) машине Тьюринга;

г) ограниченному линейному автомату;

д) МП-машине.

Список использованных источников

1 Ахо, А.В. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман; перевод с англ. И.В. Красикова и др. - М.: Вильямс, 2001. - 767 с.: ил.; 24 см. - Библиогр.: с. 742-763. - Предм. указ.: 764-767. - 5000 экз. - ISBN 5-8459-0189-8 (в пер.).

2 Власенко, А.В. Теория языков программирования и методы трансляции: учеб. пособие / А.В. Власенко, В.И. Ключко; М-во образования и науки РФ, ГОУ ВПО «Кубан. гос. технол. ун-т». - Краснодар: Изд-во КубГТУ, 2004. - 119 с.: ил.; 21 см. - Библиогр.: с. 118. - 75 экз. - ISBN 5-8333-0176-9.

3 Гавриков, М.М. Основы конструирования компиляторов: учеб. пособие / М.М. Гавриков, А.Н. Иванченко, Д.В. Гринченков; М-во общ. и проф. образования РФ, Новочеркас. гос. техн. ун-т. – Новочеркасск: НГТУ, 1997. - 80 с.: ил.; 20 см. - Библиогр.: с. 79. – 75 экз. - ISBN 5-88998-059-9.

4 Гордеев, А.В. Системное программное обеспечение: учеб. для вузов / А. Ю. Молчанов. - 3-е изд. - СПб.: Питер, 2010. - 398 с.: ил. - (Учебник для вузов). - Указ. лит.: с. 387-390. - Алф. указ.: с. 391-397. - ISBN 978-5-49807-153-4.

5 Ишакова, Е.Н. Теория языков программирования и методов трансляции: учебное пособие / Е.Н. Ишакова. – Оренбург: ИПК ГОУ ОГУ, 2007. – 137 с. - ISBN 978-5-7410-0712-9.

6 Ишакова, Е.Н. Разработка компиляторов: Методические указания к курсовой работе / Е.Н. Ишакова. - Оренбург: ГОУ ОГУ, 2005. – 50 с.

7 Карпов, В.Э. Классическая теория компиляторов: учеб. пособие / В.Э. Карпов; М-во образования РФ, Моск. гос. ин-т электрон. и математики (техн. ун-т). - М.: МГИЭМ, 2002. - 78 с.: ил.; 20 см. - Библиогр.: с. 78. - 150 экз. - ISBN 5-230-16344-5.

8 Компаниец, Р.И. Системное программирование: основы построения трансляторов: учеб. пособие для высших и средних учебных заведений / Р.И.

Компаниец, Е.В. Маньков, Н.Е. Филатов. - СПб.: Корона принт, 2000. - 254, [1] с.: ил.; 23 см. - Библиогр.: с. 255. - 3000 экз. - ISBN 5-7931-0124-1.

9 Мозговой, М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход / М.В. Мозговой. – СПб.: Наука и техника, 2006. - 320 с.: ил.; 24 см. - 3000 экз. - ISBN 5-94387-224-8.

10 Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц; пер. с англ. - СПб.: Питер принт, 2002. - 688 с.: ил.; 24 см. - Библиогр.: с. 669-674. - Алф. указ.: с. 675-688. - Загл. и авт. ориг.: Programming languages / Terrence W. Pratt, Marvin V. Zelkowitz. - 4000 экз. - ISBN 5-318-00189-0 (в пер.).

11 Рейуорд-Смит, В. Теория формальных языков. Вводный курс / В. Рейуорд-Смит; пер. с англ. Б.А. Кузьмина; под ред. И.Г. Шестакова. - М.: Радио и связь, 1988. - 127, [2] с.: ил.; 21 см. - Перевод изд.: A first course in formal language theory / V.J. Rayward Smith (Oxford etc.). - 30000 экз. - ISBN 5-256-00159-0.

12 Серебряков, В.А. Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин. - М.: Эдиториал УРСС, 2001. - 221, [1] с.: ил.; 20 см. - Библиогр. в конце кн. – 1000 экз. - ISBN 5-8360-0242-8.

13 Соколов, А.П. Системы программирования: теория, методы, алгоритмы: учеб. пособие для студентов, обучающихся по направлению 654600 - Информатика и вычисл. техника / А.П. Соколов. – М.: Финансы и статистика, 2004. – 319, [1] с.: ил.; 21 см. - Библиогр.: с. 309-310. – Предм. указ.: с. 313-320. – 4000 экз. – ISBN 5-279-02770-7.

14 Теория и реализация языков программирования: учебное пособие по курсу теории и реализации языков программирования / В.А. Серебряков, М.П. Галочкин, Д.Р. Гончар, М.Г. Фуругян. – М.: МЗ-Пресс, 2006. - 348, [1] с.: ил.; 20 см. - Библиогр.: с. 347-249. – 2000 экз. - ISBN 5-94073-094-9.

15 Хантер, Р. Основные концепции компиляторов: / Р. Хантер; пер. с англ. - М.: Вильямс, 2002. - 252 с.: ил.; 21 см. - Библиогр.: с. 247-248. - Предм. указ.: с. 249-252. - 3000 экз. - ISBN 5-8459-0360-2.

Приложение А
(обязательное)

Укрупненная схема алгоритма программного средства

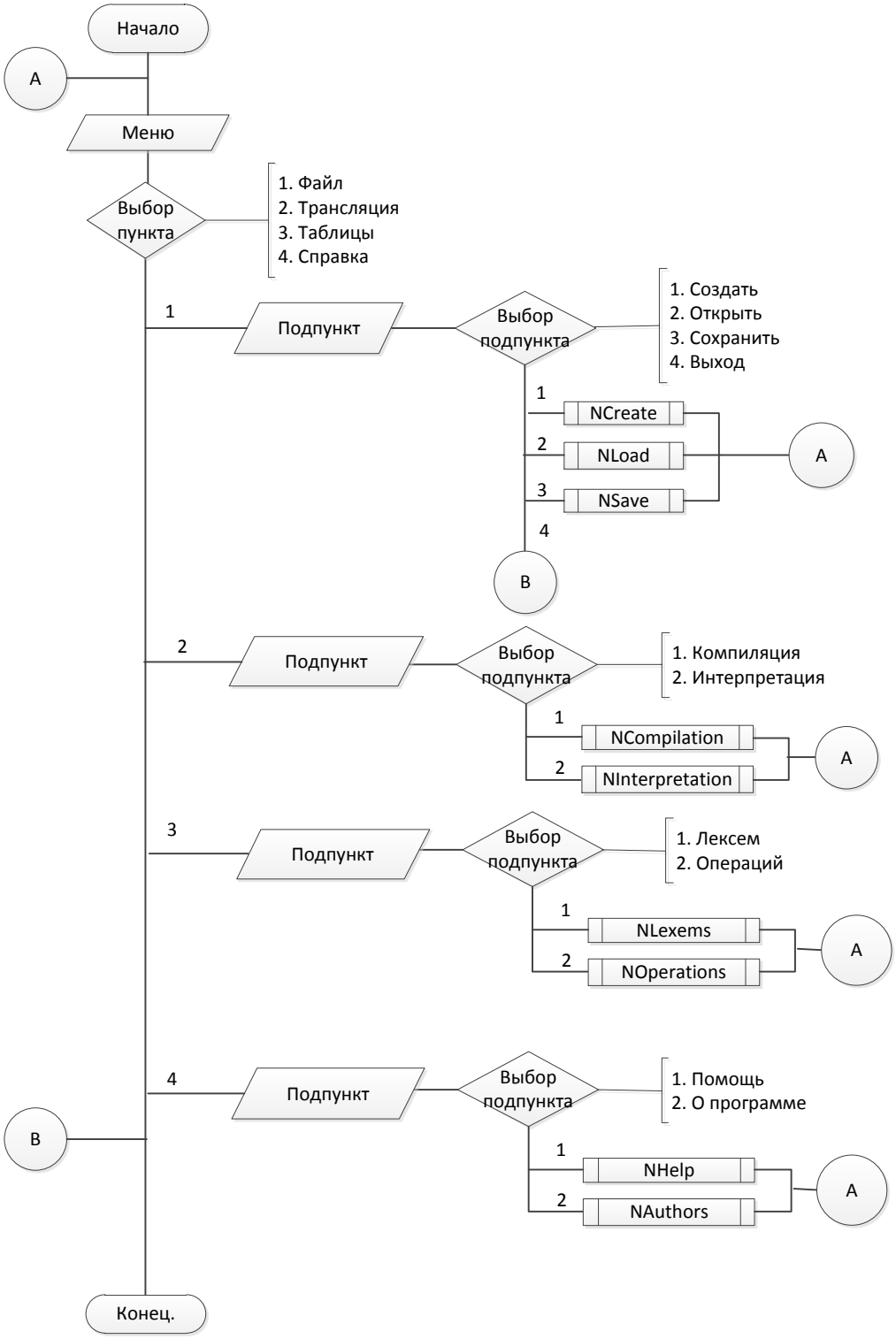


Рисунок А.1 – Укрупненная схема алгоритма программного средства

Приложение Б
(обязательное)

Контрольный пример

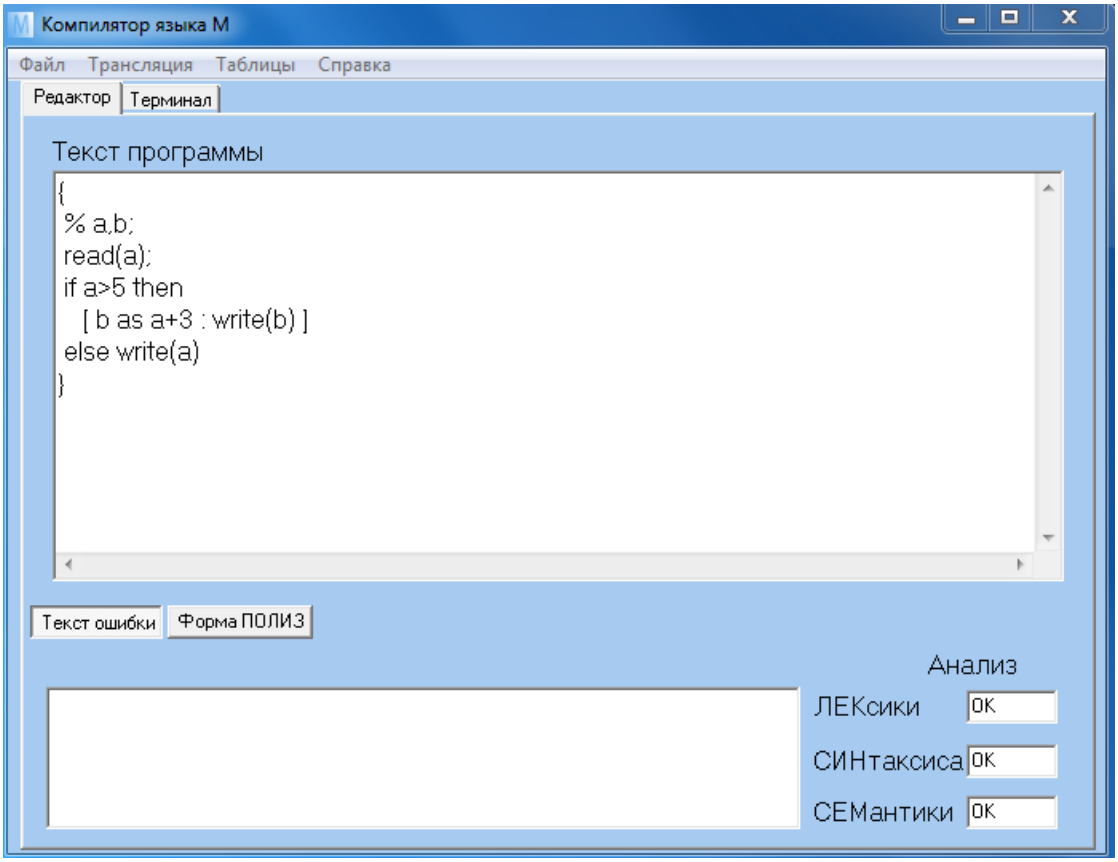


Рисунок Б.1 – Ввод исходных данных распознавателя

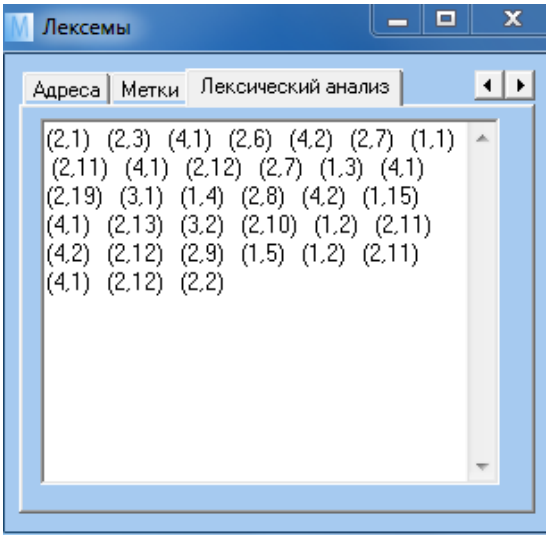


Рисунок Б.2 – Выходные данные лексического анализатора

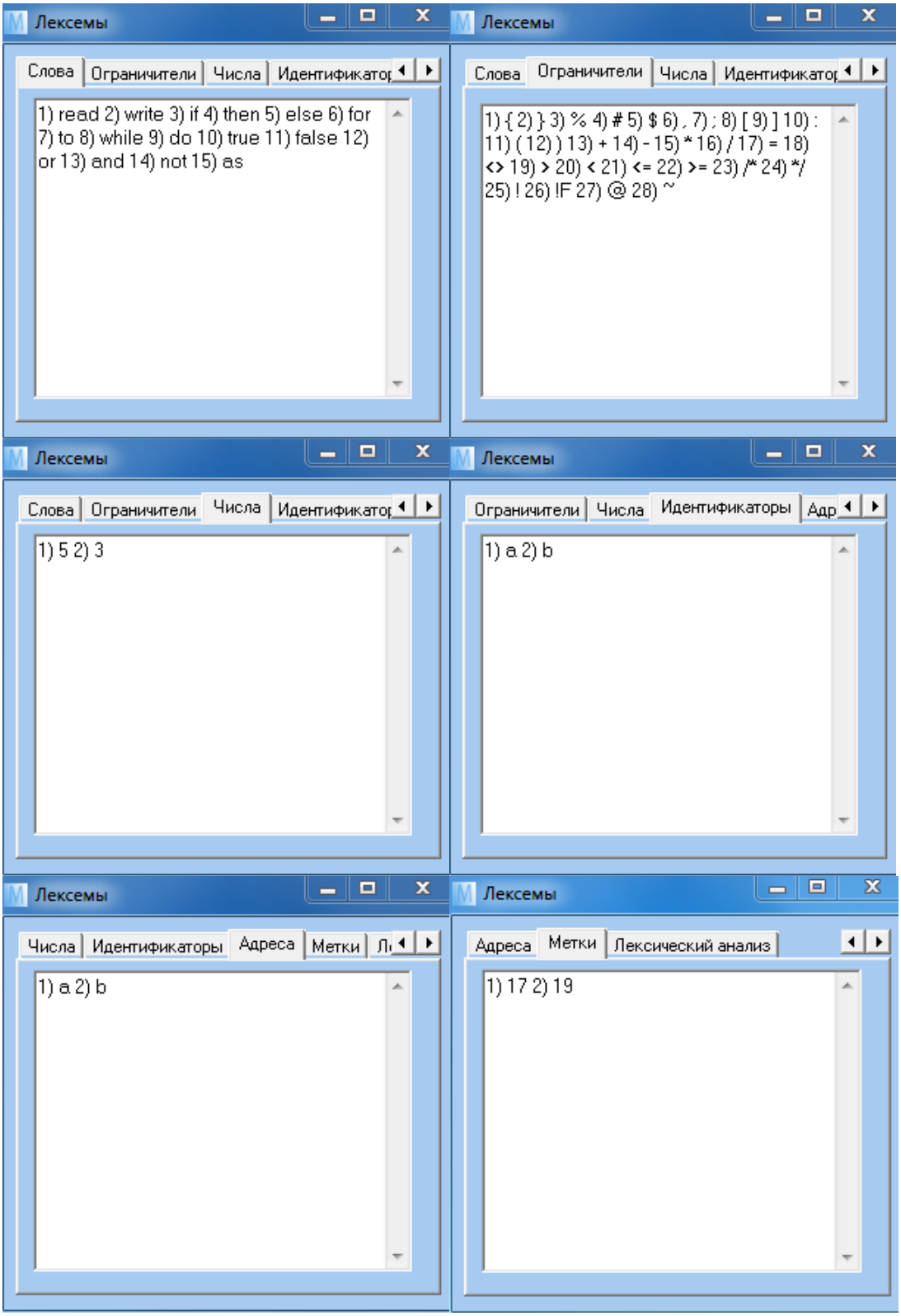


Рисунок Б.3 – Таблицы лексем

Приложение В
(обязательное)

Сообщения об ошибках

Лексический анализатор

{While true do write(0);

Ошибка #0010: Неожиданный конец файла

{a as .E22;}

Ошибка #0013: Найдено неправильное число

Синтаксический анализатор

Dim a integer;}

Ошибка #0051: Не найдена точка входа в программу

{dim a, integer;}

Ошибка #0052: Неправильное объявление переменной

{if true then [write(0) :write(1);}

Ошибка #0067: Нарушен баланс скобок составного оператора

Семантический анализатор

{Dim a integer; a as 4 / 2.0;}

Ошибка #0073: Несоответствие типов левой и правой части оператора присваивания

{if (4 / 2) then write(1);}

Ошибка #0074: Тип условия условного оператора не является "Boolean"

{ % a; for a as 0.1 to 1 do write(1)}

Ошибка #0075: Счетчик оператора for не является целым

Приложение Г

(обязательное)

Фрагмент текста программы

```
#define fmax(a,b) double( (a)>=(b)?a:b )
#include <fstream>
#include <iostream>
#include <cmath>
#include <string>
using namespace std;
// состояния диаграммы
enum sost {ER, // ошибка
           V,  // выход
           H,  // начало
           I,  // идентификатор
           C1,C2,C3, // комментарий
           M1, // < <= <>
           M2, // > >=
           N2, // двоичное число
           N8, // восьмеричное число
           N10,// десятичное число
           N16,// шестнадцатеричное число
           B,  // 'B' или 'b'
           O,  // 'O' или 'o'
           D,  // 'D' или 'd'
           HX, // 'H' или 'h'
           E11,// 'E' или 'e'
           E12,E13,E21,E22, // порядок числа
           ZN, // знак порядка
           P1, // точка
           P2, // дробная часть
           OG};// ограничитель
// таблицы лексем
enum e_tables {
             TW=1, // таблица служебных слов
             TL=2, // таблица ограничителей
             TN=3, // таблица чисел
             TI=4};// таблица идентификаторов
const int s_stack=1000,n_tables=6;
char CH,stack[s_stack];
string tbl[6][s_stack];
// внешние файлы: файл слов,
// файл ограничителей, результирующий файл
// лексического анализа
fstream fcin("input.txt"),f_slova("tbl_slova.txt"),
        f_ogran("tbl_ogranichitel.txt");
ofstream  f_out("result_tbl.txt");
int u_stack,        // указатель на вершину стека
    z, // размеры таблиц
s_tbl0,s_tbl1,s_tbl2,s_tbl3,s_tbl4,s_tbl5,
s_res_tbl,                  // размер таблицы ЛА
cur_res_tbl, // текущий элемент таблицы ЛА
n_m_err,                    // количество ошибок
bool scanner(){
 sost CS;
 gc();CS=H;
 do{
ffree, // текущий номер свободного элемента
 old, // количество уже прочитанных слов окна диалога

go_enter, // флаг готовности ввода в диалоговое окно
str_numb,str_otstup,
 res_tbl[s_stack][2],                  // массив ЛА
stroka[s_stack];
int s_TOP;          // размер таблицы операций
double m,enter_numb;
bool uspeh,         // флаг-результат завершения
ERR,SemERR, // флаги ошибок этапов распознавания
ready_enter;
struct TLEX{ // структура, описывающая лексему
        int n_tabl, // номер таблицы
        leksema;   // номер в таблице
};typedef struct TLEX tlex;
 static tlex lex,      // обрабатываемая лексема СинА
 Pol_stack[s_stack]; // стек
// структура таблицы идентификаторов
 struct Ttabid{       // таблица идентификаторов
          char id[s_stack], typid[s_stack];
          bool descrid,nb;
          double nd;
          int ni;
};  typedef Ttabid tabid;
 static tabid mTI[s_stack];
 struct tTOP{   // структура таблицы операций
            char      op[s_stack],      t1[s_stack],
            t2[s_stack], t_res[s_stack];
 };
 struct tTOP TOP[s_stack];
 // структура элементов
 struct t2types{
              int i; // значение элемента типа % или $
              double  d;  // значение  элемента типа #
              char type; // тип элемента
 }stackI[s_stack];
 // структура номера ошибок
 struct tError{
              int number, str_numb;
 }m_err[100]; // массив ошибок программы

 switch(CS){
 case H:{
 while( (CH==' ' || CH=='\n' || CH=='\r' )&& !fcin.eof() ){
```

```
            if(CH=='\n' || CH=='\r')
              str_numb++;
            gc();
        }
        if(fcin.eof() ){
            CS=ER;
            SynA::err_proc(3);
        }
        if(let()){
            nill();add();
            gc();CS=I;
        }
        else if(CH=='0' || CH=='1'){
                nill();CS=N2;
                add();gc();
            }
            else if (CH>='2' && CH<='7'){
                    nill();CS=N8;
                    add();gc();
                }
                else if (CH>='8' && CH<='9'){
                        nill();CS=N10;
                        add();gc();
                    }
                    else if(CH=='.'){
                            nill(); add(); gc();
                            CS=P1;
                        }
                        else if(CH=='/'){
                                gc(); CS=C1;
                            }
                            else if(CH=='<'){
                                    gc();CS=M1;
                                }
                                else if(CH=='>'){
                                        gc();CS=M2;
                                    }
                                    else       if(CH=='}'){
out(2,2);CS=V;
                                    }
                                        else CS=OG;
        break;
        }
        case I:{
            while(let() || digit()){
                add();gc();
            }
            look(TW);
            if(z!=0){
                out(1,z);CS=H;
            }
            else{ put(TI); out(4,z);

              strcpy(mTI[z-1].id,stack); CS=H;
            }
            break;
        }
        case N2:{
            while(CH=='0' || CH=='1'){
                add(); gc();
            }
            if(CH>='2' && CH<='7')
                CS=N8;
            else if(CH=='8' || CH=='9')
                    CS=N10;
                else  if(CH=='A'  ||  CH=='a'  ||  CH=='C'  ||
CH=='c' || CH=='F' || CH=='f')
                        CS=N16;
                    else if(CH=='E' || CH=='e'){
                            add(); gc(); CS=E11;
                        }
                        else if(CH=='D' || CH=='d'){
                                add; gc(); CS=D;
                            }
                            else if(CH=='O' || CH=='o')
                                    CS=O;
                                else if(CH=='H' || CH=='h'){
                                        gc(); CS=HX;
                                    }
                                    else if(CH=='.'){
                                            add(); gc(); CS=P1;
                                        }
                                        else if(CH=='B' || CH=='b'){
                                                add();gc();
                                                CS=B;
                                            }
                                            else if(let() )
                                                    CS=ER;
                                                else CS=N10;
            break;
        }
        case N8:{
            while(CH>='2' && CH<='7'){
                    add(); gc();
            }
            if(CH=='8' || CH=='9')
                CS=N10;
            else if(CH=='A' || CH=='a' || CH=='B' ||
                    CH=='b' || CH=='C' || CH=='c' ||
                    CH=='F' || CH=='f')
                        CS=N16;
                else if(CH=='E' || CH=='e'){
                        add(); gc(); CS=E11;
                    }
                    else if(CH=='D' || CH=='d'){
                            add(); gc(); CS=D;
                        }
                        else if(CH=='H' || CH=='h'){
                                gc(); CS=HX;
                        } else if(CH=='.')
```