Разработка компилятора модельного языка

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К КУРСОВОЙ РАБОТЕ

В методических указаниях содержатся материалы, необходимые для самостоятельной подготовки студентов к выполнению курсовой работы по разработке компиляторов. В описание курсовой рабы включены цель работы, порядок ее выполнения, рассмотрены теоретические вопросы, связанные с реализацией поставленных задач, приведена необходимая литература и контрольные вопросы для самопроверки. В приложениях представлены правила оформления результатов курсовой работы.

Методические указания предназначены для выполнения курсовой работы по дисциплине «Теория языков программирования и методов трансляции» для студентов специальности 220400 – «Программное обеспечение вычислительной техники и автоматизированных систем».

Содержание

Введение 4
1 Тема и цель курсовой работы 5
2 Основы теории разработки компиляторов 5
2.1 Методы описания синтаксиса языка программирования 5
2.2 Общая структура компилятора 13
2.3 Лексический анализатор программы 14
2.4 Синтаксический анализатор программы 19
2.5 Семантический анализатор программы 24
2.6 Генерация внутреннего представления программы 29
2.7 Интерпретатор программы 32
3 Постановка задачи к курсовой работе 35
4 Требования к содержанию курсовой работы 36
5 Варианты индивидуальных заданий 37
6 Контрольные вопросы для самопроверки 42
Список использованных источников 43

Введение

Предлагаемый материал посвящен основам классической теории компиляторов – одной из важнейших составных частей системного программного обеспечения.

Несмотря на более чем полувековую историю вычислительной техники, формально годом рождения теории компиляторов можно считать 1957, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом. Лишь появление теории формальных языков и строгих математических моделей позволило перейти от «творчества» к «науке». Именно благодаря этому, стало возможным появление сотен новых языков программирования.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их компиляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью существующих компиляторов. Поэтому основы теории языков и формальных грамматик, а также практические методы разработки компиляторов лежат в фундаменте инженерного образования по информатике и вычислительной технике.

Предлагаемый материал затрагивает основы методов разработки компиляторов и содержит сведения, необходимые для изучения логики их функционирования, используемого математического аппарата (теории формальных языков и формальных грамматик, метаязыков). В методических указаниях содержатся материалы, необходимые для самостоятельной подготовки студентов к выполнению курсовой работы. В описание курсовой рабы включены цель работы, порядок ее выполнения, рассмотрены теоретические вопросы, связанные с реализацией поставленных задач, приведена необходимая литература и контрольные вопросы для самопроверки. В приложениях представлены правила оформления результатов курсовой работы.

1 Тема и цель курсовой работы

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

Цель курсовой работы:

- закрепление теоретических знаний в области теории формальных языков, грамматик, автоматов и методов трансляции;
- формирование практических умений и навыков разработки собственного компилятора 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 разработки компиляторов

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 $\alpha \to \beta_1, \alpha \to \beta_2, \text{K}, \alpha \to \beta_n$ используется сокращенная форма записи $\alpha \to \beta_1 \mid \beta_2 \mid \text{K} \mid \beta_n$.

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

$P \rightarrow program\ D2\ B.$
$D2 \rightarrow var\ D1$
$D1 \rightarrow D \mid D1;\ D$
$D \rightarrow I1:\ int \mid I1:\ bool$
$I1 \rightarrow I \mid I1,\ I$
$B \rightarrow begin\ S1\ end$
$S1 \rightarrow S \mid S1;\ S$
$S \rightarrow begin\ S1\ end \mid if\ E\ then\ S\ else\ S \mid while\ E\ do\ S \mid read(I) \mid write(E)$
$E \rightarrow E1 \mid E1=E1 \mid E1>E1 \mid E1<E1$
$El \rightarrow T \mid T+E1 \mid T-E1 \mid T \vee El$
$T \rightarrow F \mid F*T \mid F/T \mid F \wedge T$
$F \rightarrow I \mid N \mid L \mid \neg F \mid (E)$
$L \rightarrow true \mid false$
$I \rightarrow C \mid IC \mid IR$
$N \rightarrow R \mid NR$
$C \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
$R \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

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*
<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*
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<идентификатор> ::= <буква> { <буква> | <цифра> }
<число> ::= {/< цифра> /}
<ключевое_слово> ::= *program* | *var* | *begin* | *end* | *int* | *bool* | *read* | *write* | *if* | *then* | *else* | *while* | *do* | *true* | *false*
<разделитель> ::= (|) | , | ; | : | := | . | { | } |+ | - | * | / | ∨ | ∧ | ¬ | = | < | >
<программа> ::= *program* <описание> ; <тело>.
<описание> ::= *var* <идентификатор> {, <идентификатор>}: (*int* | *bool*)
<тело> ::= *begin* {<оператор>; } *end*
<оператор> ::= <присваивания> | <условный> | <цикла> | <составной> | <ввода> | <вывода>
<присваивания> ::= <идентификатор> := <выражение>
<условный> ::= *if* <выражение> *then* <оператор> *else* <оператор>
<цикла> ::= *while* <выражение> *do* <оператор>
<составной>:: = *begin* {<оператор> ;} *end*
<ввода>:: = *read*(<идентификатор>)
<вывода>:: = *write*(<выражение>)
<выражение>:: = <сумма> | <сумма> (= | < | >) <сумма>
<сумма> ::= <произведение> { (+ | - | ∨) <произведение>}
<произведение>:: = <множитель> { (* | / | ∧) <множитель>}
<множитель>:: = <идентификатор> | <число> | <логическая_константа> | ¬ <множитель> | (<выражение>)
<логическая_константа>:: = *true* | *false*

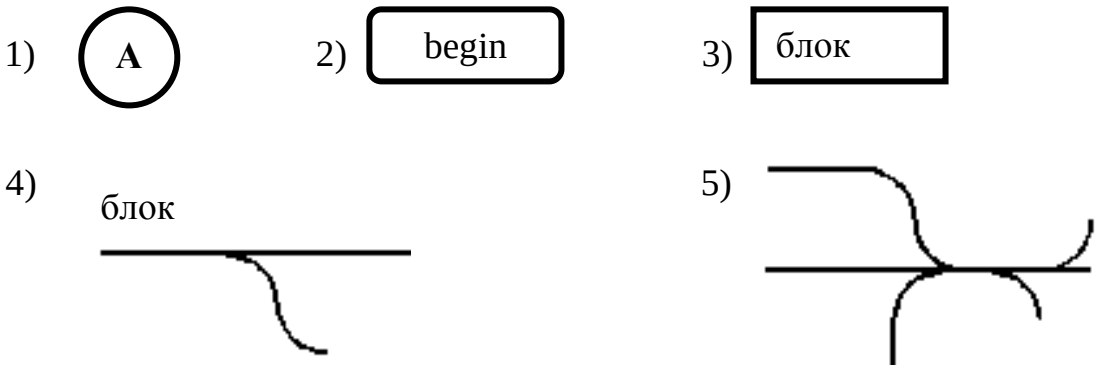
Диаграммы Вирта

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.

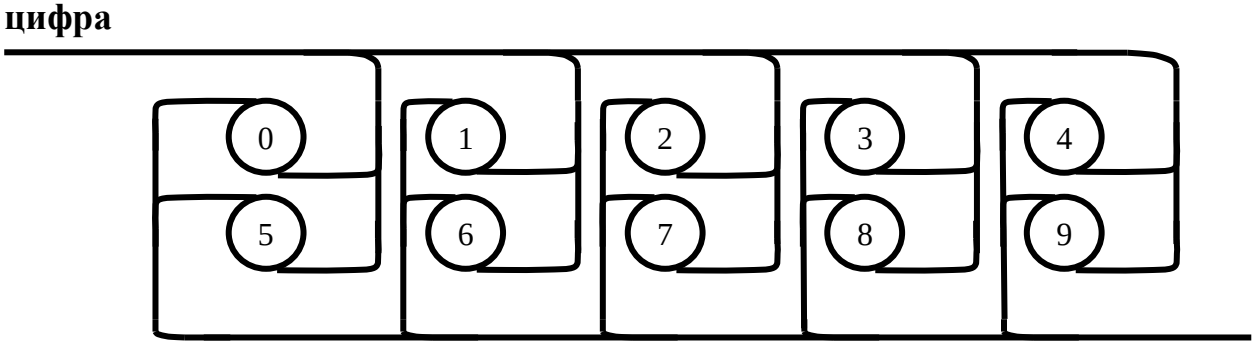


Рисунок 2.2 – Синтаксические правила модельного языка *M*

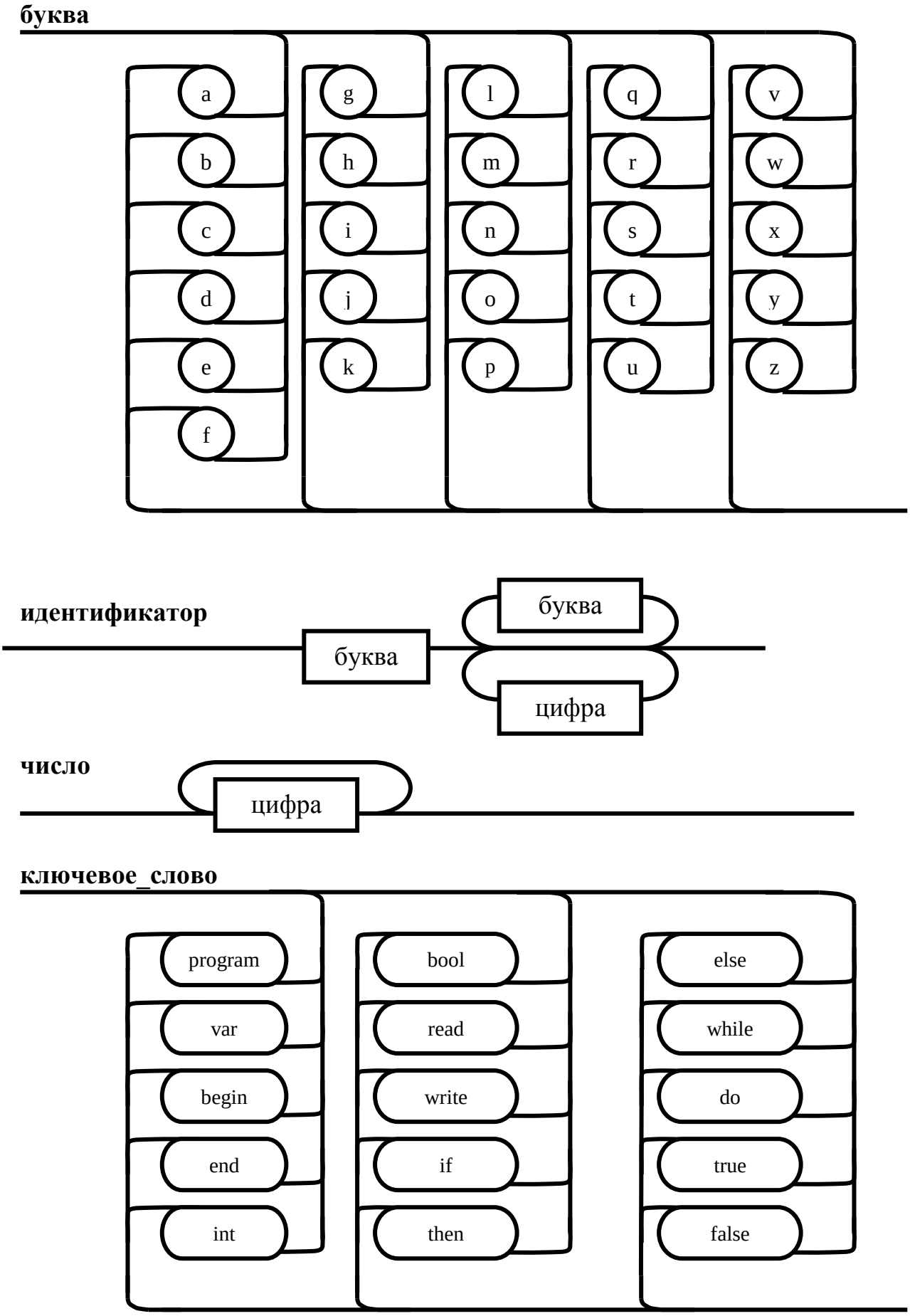


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 2

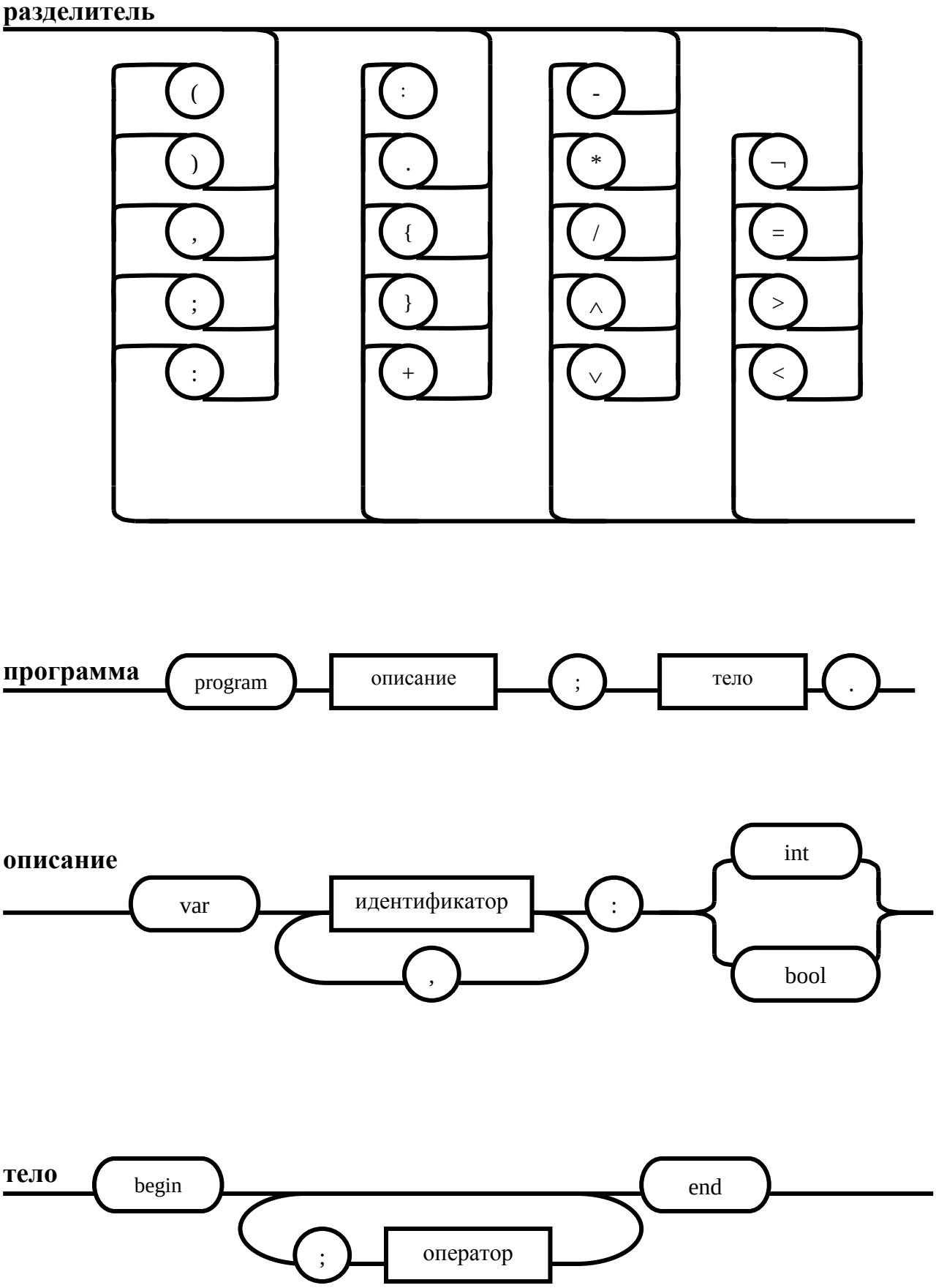


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 3

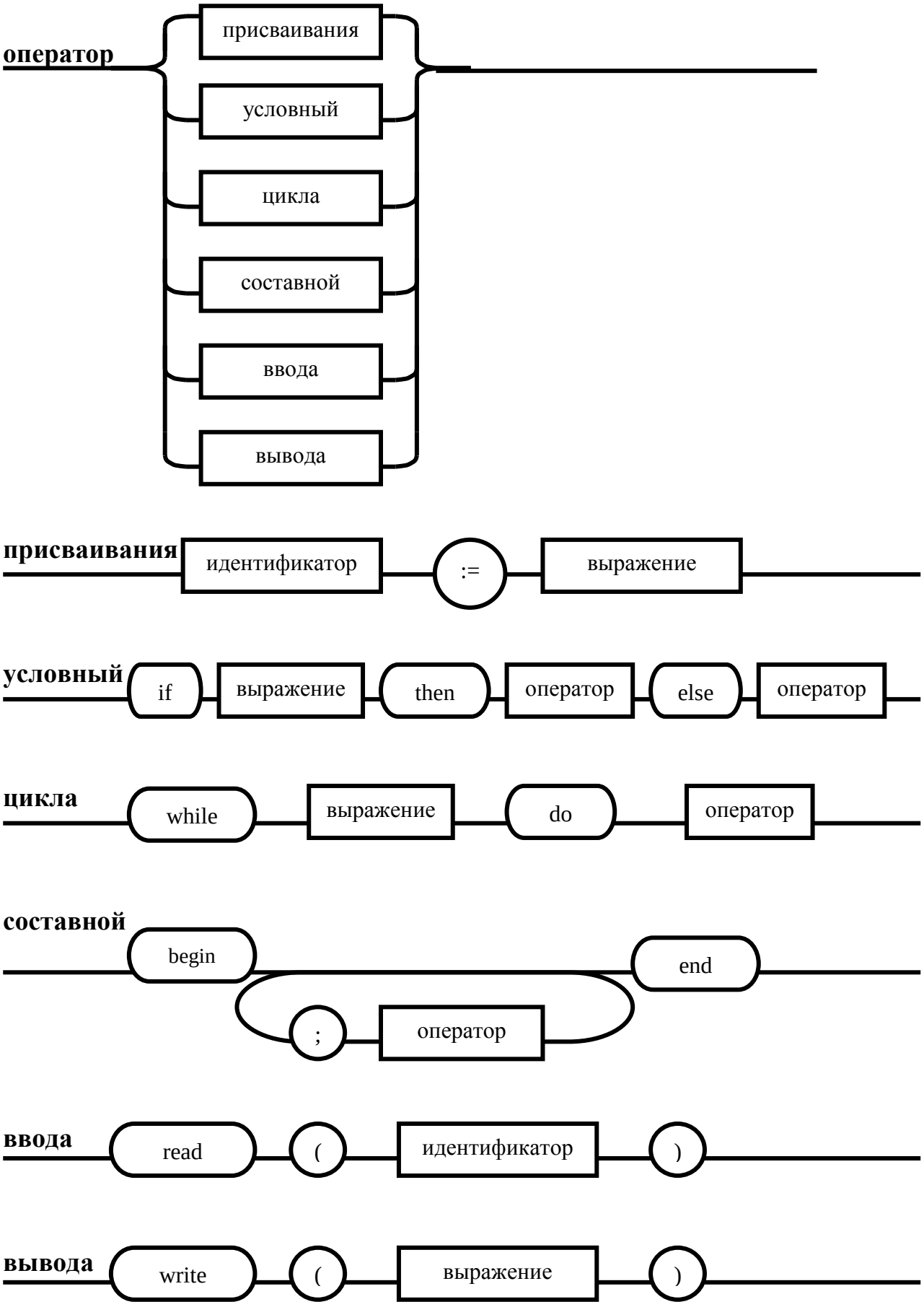


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 4

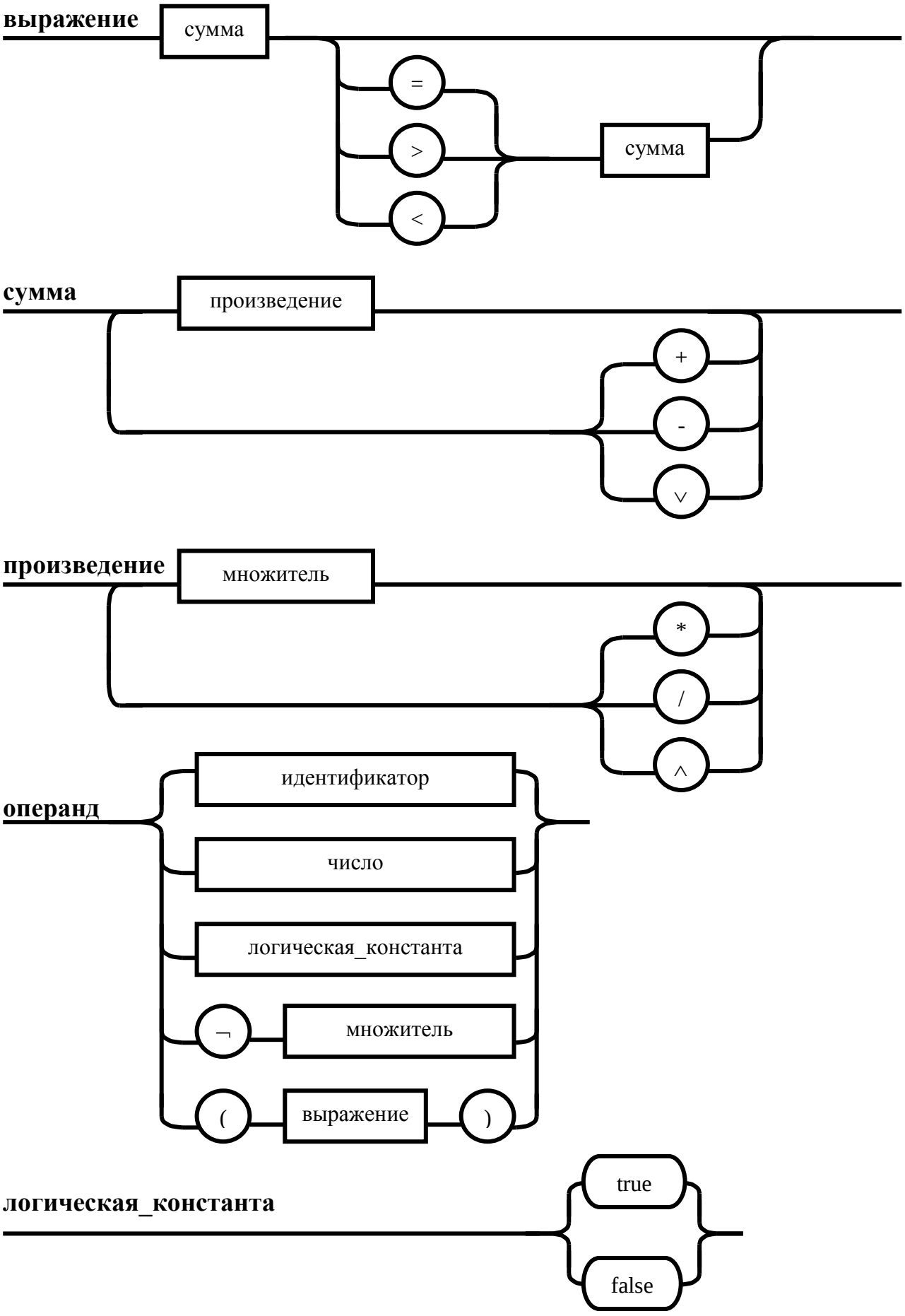


Рисунок 2.2 – Синтаксические правила модельного языка *M*, лист 5

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

```
program
var k, n, sum: int;
begin
  read(n);
  sum:= 0;
  i:=1;
 while i<=n do
     begin
         read(k);
         sum:=sum+k;
         k:=k+1
     end;
 write(sum/n)
end.
```

2.2 Общая структура компилятора

Определение 2.2. Компилятор – это программа, которая осуществляет перевод исходной программы на входном языке в эквивалентную ей объектную программу на языке машинных команд или языке ассемблере.

Основные функции компилятора:

1) проверка исходной цепочки символов на принадлежность к входному языку;

2) генерация выходной цепочки символов на языке машинных команд или ассемблере.

Процесс компиляции состоит из двух основных этапов: синтеза и анализа. На этапе анализа выполняется распознавание текста исходной программы и заполнение таблиц идентификаторов. Результатом этапа служит некоторое внутреннее представление программы, понятное компилятору.

На этапе синтеза на основании внутреннего представления программы и информации, содержащейся в таблице идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код.

Данные этапы состоят из более мелких стадий, называемых фазами. Состав фаз и их взаимодействие зависит от конкретной реализации компилятора. Но в том или ином виде в каждом компиляторе выделяются следующие фазы:

1) лексический анализ;
2) синтаксический анализ;
3) семантический анализ;
4) подготовка к генерации кода;
5) генерация кода.

Определение 2.3. Процесс последовательного чтения компилятором данных из внешней памяти, их обработки и помещения результатов во внешнюю память, называется проходом компилятора.

По количеству проходов выделяют одно-, двух-, трех- и многопроходные компиляторы. В данном пособии предлагается схема разработки трехпроходного компилятора, в котором первый проход – лексический анализ, второй - синтаксический, семантический анализ и генерация внутреннего представления программы, третий – интерпретация программы.

Общая схема работы компилятора представлена на рисунке 2.3.

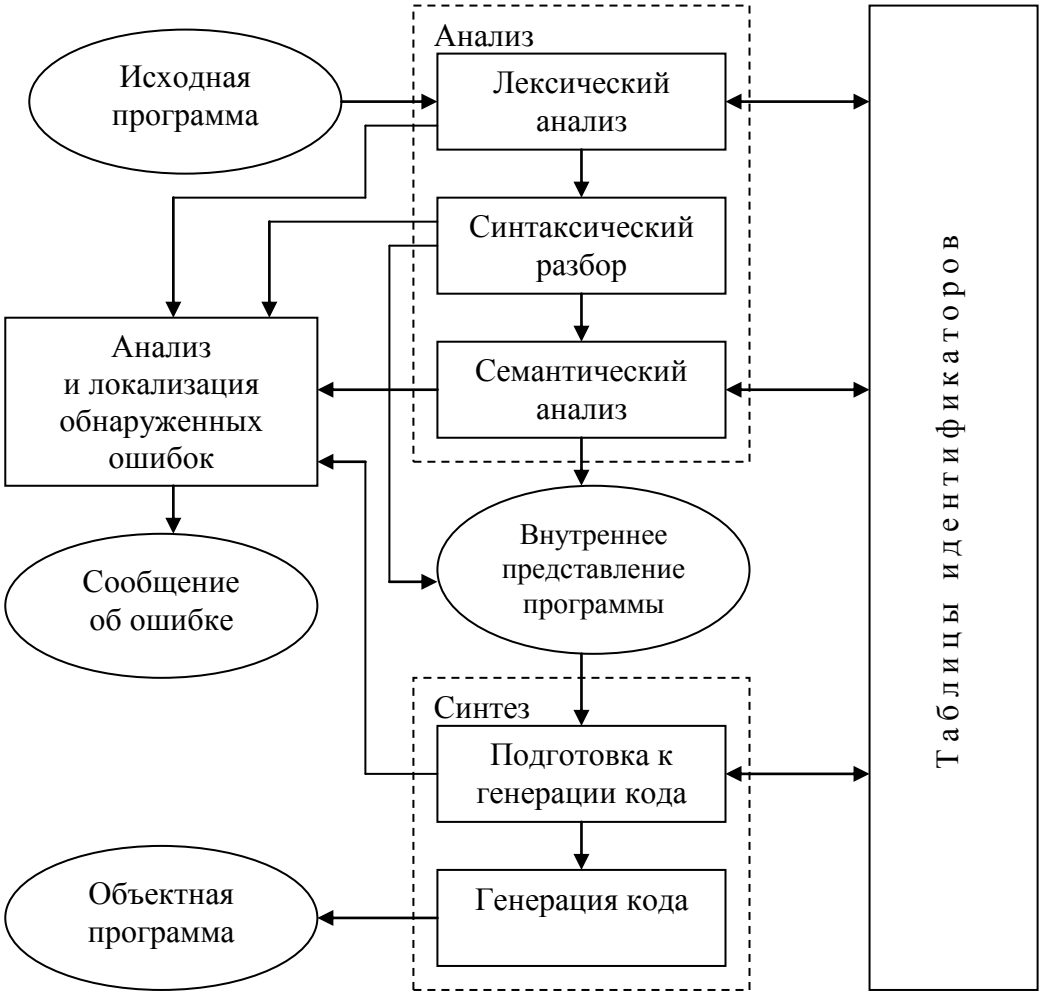


Рисунок 2.3 – Общая схема работы компилятора

2.3 Лексический анализатор программы

Определение 2.4. Лексический анализатор (ЛА) – это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются 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 необязательный этап компиляции, но желательный 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 процедурных языках 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) *program*; 2) *var*; 3) *int*; 4) *bool*; 5) *begin*; 6) *end*; 7) *if*; 8) *then*; 9) *else*; 10) *while*; 11) *do*; 12) *read*; 13) *write*; 14) *true*; 15) *false*.

Таблица ограничителей содержит:

1) . ; 2) ; ; 3) , ; 4) : ; 5) := ; 6) (; 7)) ; 8) + ; 9) - ; 10) * ; 11) / ; 12) $\vee$; 13) $\wedge$; 14) $\neg$; 15) = ; 16) > ; 17) <.

Таблицы идентификаторов и чисел формируются в ходе лексического анализа.

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

Входные данные ЛА: *program var k, sum*: *int*; *begin k*:=0;…

Выходные данные ЛА: (1, 1) (1, 2) (4, 1) (2, 3) (4, 2) (2, 4) (1, 3) (2, 2) (1, 5) (4, 1) (2, 5) (3, 1) (2, 2)…

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 $D_1, D_2, \ldots, D_m$.

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.

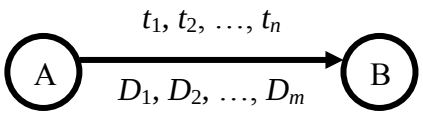


Рисунок 2.4 – Дуга ДС с действиями

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 следующие обозначения для переменных, процедур и функций.

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* - число, *C* - комментарий, *DV* – двоеточие, *O* - 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* – чисел, используемых в программе;

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

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*) – процедура 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.

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

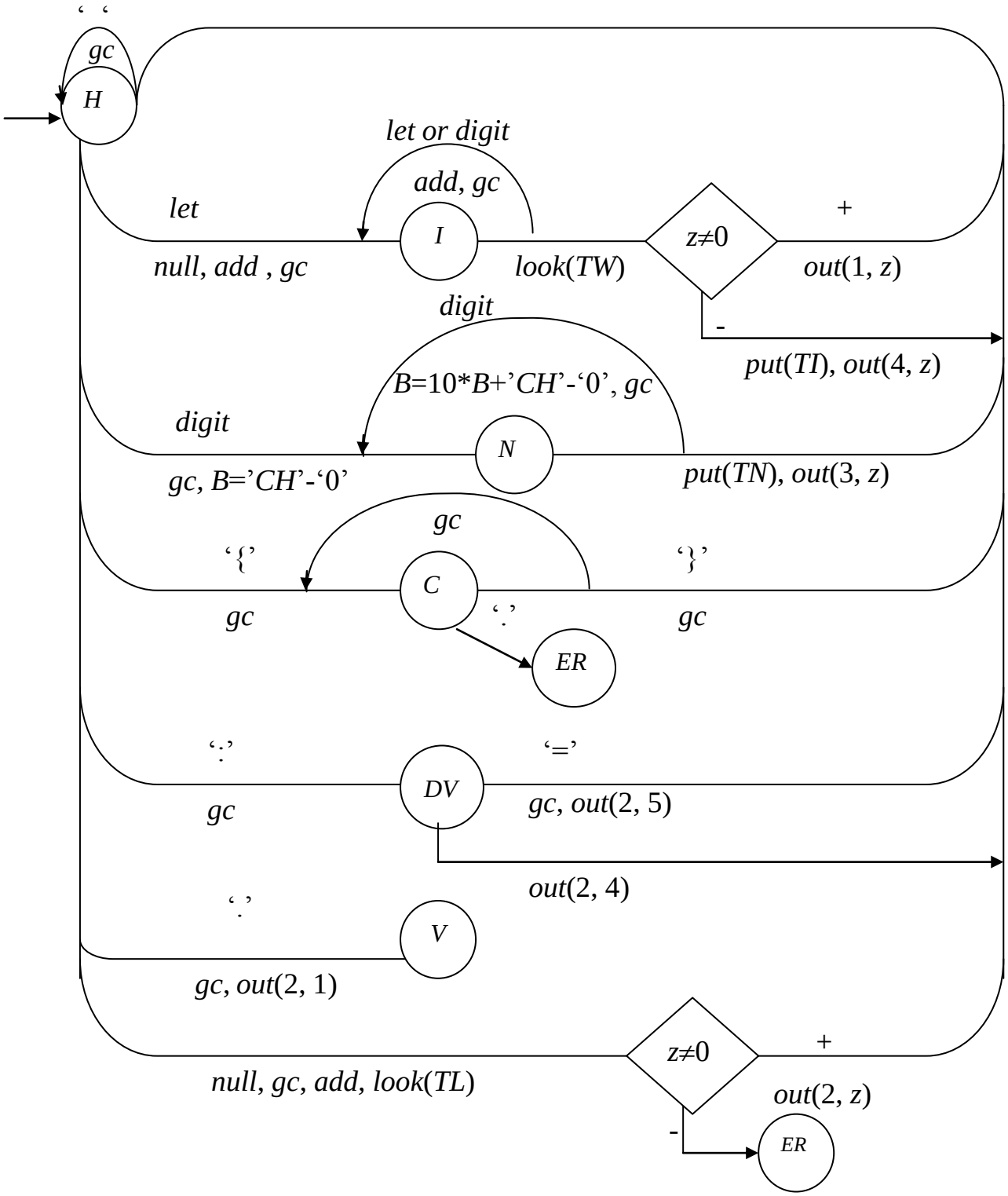


Рисунок 2.5 – Диаграмма состояний с действиями для модельного языка *M*

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

function scanner: *boolean*;
var CS: (*H*, *I*, *N*, *C*, *DV*, *O*, *V*, *ER*);
begin gc; *CS*:=*H*;

```
repeat
    case CS of
        H: if CH=' ' then gc
            else
                if let then
                    begin
                        nil; add;
                        gc; CS:= I
                    end
                else
                    if digit then
                        begin
                            B:=ord(CH)-ord('0');
                            gc; CS:= N
                        end
                    else
                        if CH= ':' then
                            begin
                                gc;
                                CS:= DV
                            end
                        else
                            if CH='.' then
                                begin
                                    out(2,1);
                                    CS:=V
                                end
                            else
                                if CH='{' then
                                    begin
                                        gc; CS:=C
                                    end
                                else CS:=O;
        I: if let or digit then
                begin
                    add; gc
                end
            else begin
                    look(TW);
                    if z<>0 then
                        begin
                            out(1,z); CS:=H
                        end
                    else begin
                            put(TI);
```

```
                        out(4,z);
                        CS:=H
                    end
            end;
        N: if digit then
            begin
                B:=10*B+ord(CH)-ord('0');
                gc
            end
          else begin
                put(TN);
                out(3,z); CS:=H
              end;
        C: if CH='}' then begin
                            gc; CS:=H
                          end
          else if CH='.' then CS:=ER else gc;
        DV: if CH='=' then begin
                             gc; out(2,5);
                             CS:=H
                           end
           else begin
                  out(2,4); CS:=H
                end;
        O: begin
            null; add; look(TL);
            if z<>0 then begin
                           gc; out(2,z);
                           CS:=H
                         end
            else CS:=ER
           end
      end {case}
    until (CS=V) or (CS=ER);
    scanner:= CS=V
end;
```

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.

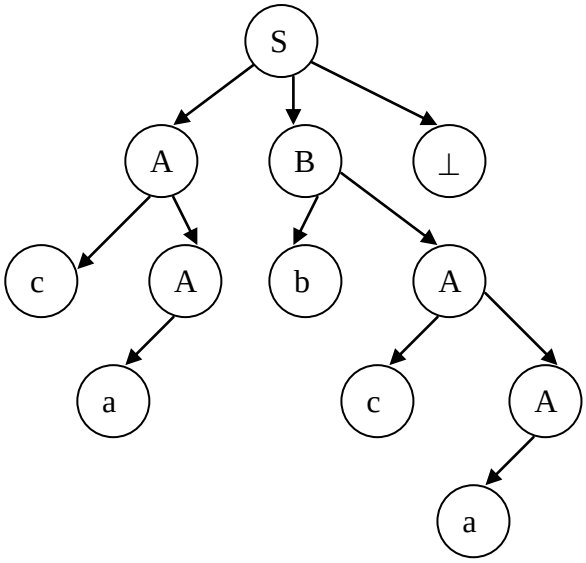


Рисунок 2.6 – Дерево нисходящего разбора цепочки $cabca\perp$

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.

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.

```
procedure S;
begin
    A; B;
    if CH<>'⊥' then ERR
end;

procedure A;
begin
    if CH='a' then gc
    else if CH='c'
        then begin
                gc; A
             end
        else Err
end;

procedure B;
begin
    if CH= 'b' then
        begin
            gc; B
        end
    else Err
end;
```

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) $A \to a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid a_n\alpha_n$, 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 (см. лабораторную работу № 4) /11/.

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 сокращенной форме $L \to a\{,a\}$.

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:

```
procedure L;
begin
    if CH<>'a' then Err else gc;
    while CH=',' do
        begin
            gc;
            if CH<>'a' then Err else gc
        end
end;
```

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*;

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

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

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

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

1) для правила $P \to$ *program D*1 *B*.

```
procedure P;
begin
    if EQ(`program`) then gl else ERR;
    D1;
```

```
    B;
    if not EQ(‘.’) then ERR
end;
```

2) для правила $D1 \rightarrow$ *var* $D\{;D\}$

```
procedure D1;
begin
    if EQ(‘var’) then gl else ERR;
    D;
    while EQ(‘;’) do
        begin
            gl; D
        end
end;
```

3) для правила $D \rightarrow I\{,I\}:(int \mid bool)$

```
procedure D;
begin
    I;
    while EQ(‘,’) do
        begin
            gl; I
        end;
    if EQ(‘:’) then gl else ERR;
    if EQ(‘int’) or EQ(‘bool’) then gl else ERR
end;
```

4) для правила $F \rightarrow I|N|L|\neg F|(E)$

```
procedure F;
begin
    if ID or NUM or EQ(‘true’) or EQ(‘false’) then gl
      else
          if EQ(‘¬’)
              then begin
                       gl; F
                   end
              else
                if EQ(‘(‘)
                    then begin
                                gl; E;
                                if EQ(‘)’) then gl else ERR
                             end
                    else ERR
end;
```

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:

```
type
   tabid = record
      id      :string;
      descrid :byte;
      typid   :string[4];
      addrid  :word
   end;
var
   TI: array[1.. n] of tabid;
```

Таблица 2.1 – Таблица идентификаторов на этапе СеА

Номер	Идентификатор	Описан	Тип	Адрес
1	*K*	1	*Int*	…
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 *D* вводится стековая переменная-массив, в которую заносится контрольное число 0. По мере успешного выполнения процедуры *I* в стек заносятся номера считываемых из файла лексем, под которыми они записаны в таблице идентификаторов. Как только при считывании лексем встречается лексема «:», 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 «*int*» или «*bool*», то попутно в таблице идентификаторов поле «тип» заполняется соответствующим типом.

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

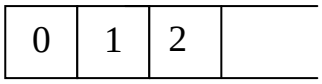


Рисунок 2.7 – Содержимое стека при выполнении процедуры *D*

Для реализации обработки описаний введем следующие обозначения переменных и процедур:

1) *LEX* – переменная, хранящая значение очередной лексемы, представляющая собой одномерный массив размером 2, т.е. для лексемы (*n, k*) *LEX*[1]=*n*, *LEX*[2]=*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*[2]);

6) *dec*(*t*) - процедура вывода всех чисел из стека и вызова процедуры *decid*(*1*, *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:

```
procedure decid (l:..; t:...);
begin
    if TI[l].descrid =1 then ERR
    else begin
            TI[l].descrid: = 1;
            TI[l].typid:= t
         end
end;

procedure dec(t: ...);
begin
    outst(l);
```

```
    while l<>0 do
        begin
            decid(l, t);
            outst(l)
        end
end;
```

Правило и процедура *D* с учетом семантических проверок принимают вид:

$D \rightarrow$ <*inst*(0)> *I* <*instl*> {, *I* <*instl*> } : (*int* <*dec*(‘*int*’)> | *bool* <*dec*(‘*bool*’)>)

```
procedure D;
begin
    inst(0);
    I;
    instl;
    while EQ(‘,’) do
        begin
            gl; I; instl
        end;
    if EQ(‘:’) then gl else ERR;
    if EQ(‘int’) then
        begin
            gl; dec(‘int’)
        end
    else
        if EQ(‘bool’)
            then
                begin
                    gl; dec(‘bool’)
                end
        else ERR
end;
```

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 оставляется только тип результата операции. В результате разбора всего выражения в стеке остается тип этого выражения.

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*) – процедура, которая по таблице операций *TOP* для операции *op* выдает тип *t* результата и типы *t*1, *t*2 операндов;

4) *checknot* – процедура проверки типа для одноместной операции «¬».

Таблица 2.2 – Фрагмент таблицы двуместных операций *TOP*

Операция	Тип 1	Тип 2	Тип результата
+	*int*	*int*	*int*
>	*int*	*int*	*bool*
…	...	...	...

Перечисленные процедуры имеют следующий вид:

```
procedure checkid;
begin
    k:=LEX[2];
    if TI[k].descrid = 0 then ERR;
    inst(TI[k].typid)
end;

procedure checkop;
begin
    outst(top2); outst(op); outst(top1);
    gettype(op, t1, t2, t);
    if (top1<>t1) or (top2<>t2) then ERR;
    inst(t)
end;

procedure checknot;
begin
    outst(t);
    if t<> bool then ERR;
    inst(t)
end;
```

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

$E \rightarrow E1$ {(> | < | =) <*instl*> *E*1 <*checkop*>}
$E1 \rightarrow T$ {(+ | - | ∨) <*instl*> *T* <*checkop*>}
$T \rightarrow F$ {(* | / | ∧) <*instl*> *F*<*checkop*>}
$F \rightarrow I$ <*checkid*>| *N*<*inst*(‘*int*’)> | *L* <*inst*(‘*bool*’)>| ¬*F* <*checknot*>|(*E*)

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

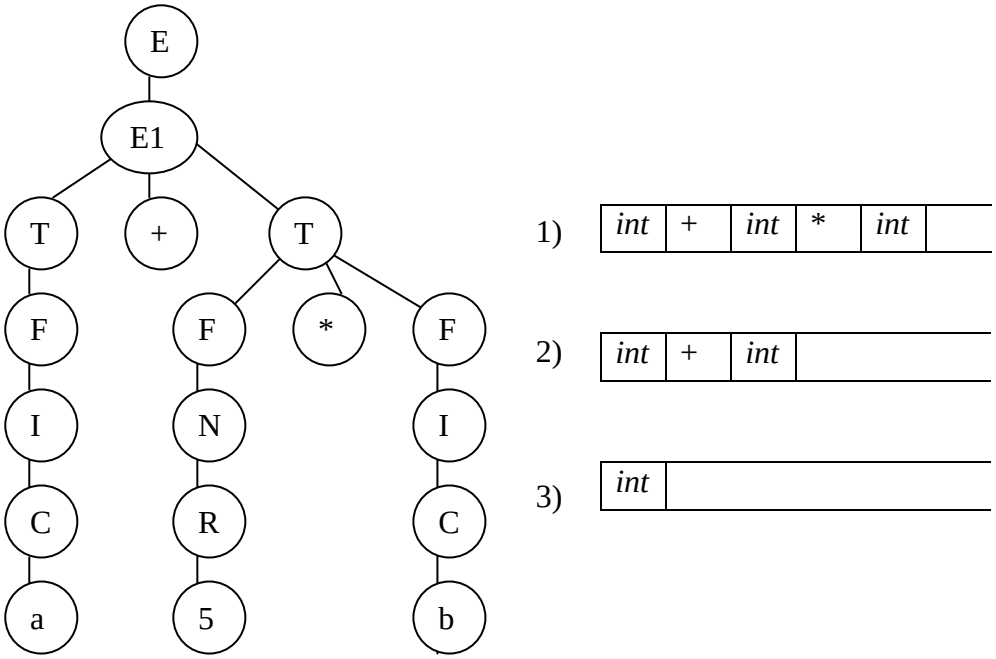


Рисунок 2.8 – Анализ выражения *a+5*b*

Проверка правильности операторов

Задачи проверки правильности операторов:

1) выяснить, все ли переменные, встречающиеся в операторах, описаны;

2) установить соответствие типов в операторе присваивания слева и справа от символа «:=»;

3) определить, является ли выражение *E* в операторах условия и цикла булевым.

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

```
procedure eqtype;
begin
    outst(t2); outst(t1);
    if t1<>t2 then ERR
end;

procedure eqbool;
begin
  outst(t);
  if t<>bool then ERR
end;
```

Правило *S* с учетом процедур СеА примет вид:

S→ *I* <*checkid*> := *E* <*eqtype*> | *if E* <*eqbool*> *then S else S*
while E <*egbool*> *do S* | *write* (*E*) | *read* (*I* <*checkid*>)

2.6 Генерация внутреннего представления программы

Результатом СиА должно быть некоторое внутреннее представление исходной цепочки лексем, которое отражает ее синтаксическую структуру. Программа в таком виде может либо транслироваться в объектный код, либо интерпретироваться.

Выделяют следующие общепринятые способы внутреннего представления программы:

1) постфиксная запись;
2) многоадресный код с явно именуемым результатом (тетрады);
3) многоадресный код с неявно именуемым результатом (триады);
4) синтаксические деревья;
5) машинные команды или ассемблерный код.

В качестве языка для представления промежуточной программы выберем постфиксную запись – ПОЛИЗ (польская инверсная запись).

Перевод в ПОЛИЗ выражений

В ПОЛИЗе операнды записаны слева направо в порядке использования. Знаки операций следуют таким образом, что знаку операции непосредственно предшествуют его операнды.

Пример 2.13. Для выражения в обычной (инфиксной записи) *a**(*b*+*c*)-(*d*-*e*)/*f* ПОЛИЗ будет иметь вид: *abc*+**de*-*f*/-.

Справедливы следующие формальные определения.

Определение 2.5. Если E является единственным операндом, то ПОЛИЗ выражения E – это этот операнд.

Определение 2.6. ПОЛИЗ выражения $E_1\ \theta\ E_2$, где θ - знак бинарной операции, E_1 и E_2 – операнды для θ, является запись $E_1' E_2' \theta$, где E_1', E_2' - ПОЛИЗ выражений E_1 и E_2 соответственно.

Определение 2.7. ПОЛИЗ выражения θE, где θ - знак унарной операции, а E – операнд θ, есть запись $E_1'\theta$, где E' - ПОЛИЗ выражения E.

Определение 2.8. ПОЛИЗ выражения (E) есть ПОЛИЗ выражения E.

Перевод в ПОЛИЗ операторов

Каждый оператор языка программирования может быть представлен как n-местная операция с семантикой, соответствующей семантике оператора.

Оператор присваивания $I:=E$ в ПОЛИЗе записывается:

$$\underline{I} E :=,$$

где «:=» - двуместная операция,

$\underline{I}$, E – операнды операции присваивания;

$\underline{I}$ – означает, что операндом операции «:=» является адрес переменной I, а не ее значение.

Пример 2.14. Оператор *x*:=*x*+9 в ПОЛИЗе имеет вид: *x x* 9 + :=.

Оператор перехода в терминах ПОЛИЗа означает, что процесс интерпретации необходимо продолжить с того элемента ПОЛИЗа, который указан как операнд операции перехода. Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они пронумерованы, начиная с единицы (например, последовательные элементы одномерного массива). Пусть ПОЛИЗ оператора, помеченного меткой *L*, начинается с номера *p*, тогда оператору безусловного перехода *goto L* в ПОЛИЗе будет соответствовать:

p!, где ! – операция выбора элемента ПОЛИЗа, номер которого равен *p*.

Условный оператор. Введем вспомогательную операцию – условный переход «по лжи» с семантикой *if* (*not B*) *then goto L*. Это двуместная операция с операндами *B* и *L*. Обозначим ее !*F*, тогда в ПОЛИЗе она будет записываться:

B p !*F*, где *p* – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой *L*.

С использованием введенной операции условный оператор *if B then* S_1 *else* S_2 в ПОЛИЗе будет записываться:

B p_1 !*F* S_1 p_2 ! S_2, где p_1 – номер элемента, с которого начинается ПОЛИЗ оператора S_2, а p_1 – оператора, следующего за условным оператором.

Пример 2.15. ПОЛИЗ оператора *if x*>0 *then x*:=*x*+8 *else x*:=*x*-3 представлен в таблице 2.3.

Таблица 2.3 – ПОЛИЗ оператора *if*

лексема	*x*	0	>	13	!*F*	*x*	*x*	8	+	:=	18	!	*x*	*x*	3	-	:=	…
номер	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Оператор цикла. С учетом введенных операций оператор цикла *while B do S* в ПОЛИЗе будет записываться:

B p_1 !*F S* p_o !, где p_o – номер элемента, с которого начинается ПОЛИЗ выражения *B*, а p_1 – оператора, следующего за данным оператором цикла.

Операторы ввода и вывода языка *M* одноместные. Пусть *R* – обозначение операции ввода, а *W* – обозначение операции вывода, тогда оператор *read*(*I*) в ПОЛИЗе запишется как *I R*, а оператор *write*(*E*) – *E W*.

Составной оператор *begin* S_1; S_2;...; S_n *end* в ПОЛИЗе записывается как S_1 S_2... S_n.

Пример 2.16. ПОЛИЗ оператора *while n*>3 *do begin write*(*n***n*-1); *n*:=*n*-1 *end* представлен в таблице 2.4.

Таблица 2.4 – ПОЛИЗ оператора *while*

лексема	*n*	3	>	19	!*F*	*n*	*n*	*	1	-	*W*	*n*	*n*	1	-	:=	1	!	…
номер	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Синтаксически управляемый перевод

На практике СиА, СеА и генерация внутреннего представления программы осуществляется часто одновременно. Способ построения промежуточной программы – синтаксически управляемый перевод. В его основе лежит грамматика с действиями. Параллельно с анализом исходной цепочки лексем осуществляются действия по генерации внутреннего представления программы. Для этого грамматика дополняется вызовами соответствующих процедур.

Пример 2.17. Составим процедуры перевода в ПОЛИЗ программы на *M* языке.

ПОЛИЗ представляет собой массив, каждый элемент которого является парой вида (*n*, *k*), где *n* – номер таблицы лексем, *k* – номер лексемы в таблице. Расширяем набор лексем:

1) в таблицу ограничителей добавляем новые операции ! (18), !*F* (19), *R* (20), *W* (21);

2) для ссылок на номера элементов ПОЛИЗа введем нулевую таблицу лексем, т.е. пара (0, *p*) - это лексема, обозначающая *p*-ый элемент в ПОЛИЗе;

3) чтобы различать операнды-переменные и операнды-адреса переменных, обозначим переменные как четвертую таблицу лексем, а адреса – пятую.

Введем следующие обозначения переменных и процедур:

1) *P* – переменная–массив, в который размещается генерируемая программа;

2) *free* – переменная, хранящая номер первого свободного элемента в массиве *P*;

3) *LEX* – переменная, хранящая очередную лексему;

4) *put_lex*(*LEX*) – запись очередной лексемы в массив *P*, т.е. *P*[*free*]:=*LEX* и *free*:=*free*+1;

5) *put_l* – запись текущей лексемы в массив *P*;

6) *put_l5* – запись текущей лексемы в массив *P* с изменением четвертого класса лексем на пятый;

7) *put_op* - запись в массив *P* знака операции, считанного процедурой *checkop*;

8) *make*(*k*) - процедура, формирующая лексему-метку (0, *k*).

Правила, описывающие выражения языка *M*, с учетом действий перевода в ПОЛИЗ принимают вид.

$E \to E1$ {(> | < | =) <*instl*> *E*1 <*checkop*; *put_op* >}
$E1 \to T$ {(+ | - | $\vee$) <*instl*> *T* <*checkop*; *put_op* >}
$T \to F$ {(* | / | $\wedge$) <*instl*> *F*<*checkop*; *put_op* >}
$F \to I$ <*checkid*; *put_l*> | *N* <*inst*(‘*int*’); *put_l*> | *L* <*inst*(‘*bool*’); *put_l*>|
$\neg F$ <*checknot*; *put_lex*(‘$\neg$’)>| (*E*)

Оператор присваивания, дополненный действиями, примет вид:

$S \to I$ <*checkid*; *put_l*5> := *E* <*eqtype*; *put_lex*(‘:=’)>

При генерации ПОЛИЗа выражений и оператора присваивания элементы массива *P* записываются последовательно. Семантика условного оператора такова, что значения операндов для операций безусловного перехода и перехода «по лжи» в момент генерации еще не неизвестны. Поэтому необходимо запоминать номера элементов массива *P*, соответствующих этим операндам, а затем, когда станут известны их значения, заполнять пропущенное.

Правила условного оператора и оператора цикла примут вид:

S→ *if E* <*egbool*; *p*1:=*free*; *free*:=*free*+1; *put_lex*(‘!*F*’)> *then S* <*p*2:=*free*; *free*:=*free*+1; *put_lex*(‘!’); *P*[*p*1]:=*make*(*free*)> *else S* <*P*[*p*2]:=*make*(*free*)>

S→ *while* <*p*0:=*free*> *E* <*egbool*; *p*1:=*free*; *free*:=*free*+1; *put_lex*(‘*!F*’)> *do S* <*P*[*free*]:=*make*(*p0*); *put_lex*(‘!’); *P*[*p*1]:=*make*(*free*) >

Правила операторов ввода и вывода с учетом действий записи в ПОЛИЗ будут преобразованы следующим образом:

S→ *write* (*E* <*put_lex*(‘*W*’)>) | *read* (*I* <*checkid*; *put_l*5; *put_lex*(‘*R*’)>)

Чтобы в конце ПОЛИЗа была точка, правило *P* переписывается в виде:

P→*program D*1 *B* <*put_lex*(‘.’)>.

Таким образом, польская инверсная запись очищена от всех служебных слов, кроме *true* и *false*; от ограничителей остаются только знаки операций и знаки «:=», «.».

2.7 Интерпретатор программы

Запись программы в форме ПОЛИЗа удобна для последующей интерпретации (выполнения программы) с помощью стека. Массив ПОЛИЗа просматривается один раз слева направо, при этом:

1) если очередной элемент ПОЛИЗа является операндом, то его значение заносят в стек;

2) если очередной элемент – операция, то на «верхушке» стека находятся ее операнды, которые извлекаются из стека, над ними выполняется соответствующая операция, результат которой заносится в стек;

3) интерпретация продолжается до тех пор, пока не будет считана из ПОЛИЗа точка, стек при этом должен быть пуст.

Пример 2.18. Интерпретировать ПОЛИЗ программы, заданный таблицей 2.5 при введенном значении *a*, равном *7*.

Таблица 2.5 – ПОЛИЗ исходной программы

Лексема	*a*	*r*	*a*	5	>	17	!*F*	*b*	*a*	3	+
(*n*, *k*)	(5,1)	(2,20)	(4,1)	(3,1)	(2,16)	(0,17)	(2,19)	(5,1)	(4,1)	(3,2)	(2,8)
Номер	1	2	3	4	5	6	7	8	9	10	11

Продолжение таблицы 2.5 – ПОЛИЗ исходной программы

Лексема	:=	*b*	*W*	19	!	*a*	*W*	.
(*n*, *k*)	(2,5)	(4,1)	(2,21)	(0,19)	(2,18)	(4,1)	(2,21)	(2,1)
Номер	12	13	14	15	16	17	18	19

Процесс интерпретации программы на модельном языке *M*, записанной в форме ПОЛИЗа, показан в таблице 2.6.

Таблица 2.6 – Ход интерпретации ПОЛИЗа программы

Стек	Текущий элемент ПОЛИЗа	Операция	Таблицы переменных	
			адреса	значения
пуст	1	адрес - в стек	1) a 2) b	1) - 2) -
1	2	извлечь из стека номер элемента таблицы значений и записать по нему число 7	1) a 2) b	1) 7 2) -
пуст	3	значение - в стек	1) a 2) b	1) 7 2) -
7	4	значение - в стек	1) a 2) b	1) 7 2) -
7; 5	5	в стек – true, т.к. 7>5	1) a 2) b	1) 7 2) -
true	6	метка - в стек	1) a 2) b	1) 7 2) -
true; 6	7	переход, к следующему элементу ПОЛИЗа, т.к. условие истинно	1) a 2) b	1) 7 2) -
пуст	8	адрес - в стек	1) a 2) b	1) 7 2) -
2	9	значение переменной - в стек	1) a 2) b	1) 7 2) -
2; 7	10	число - в стек	1) a 2) b	1) 7 2) -
2; 7; 3	11	извлечь из стека 3 и 7 и поместить в стек их сумму	1) a 2) b	1) 7 2) -
2; 10	12	присвоить второму элементу таблицы значений число 10	1) a 2) b	1) 7 2) 10
пуст	13	значение – в стек	1) a 2) b	1) 7 2) 10
10	14	вывести на экран число из «верхушки стека»	1) a 2) b	1) 7 2) 10

Продолжение таблицы 2.6 – Ход интерпретации ПОЛИЗа программы

Стек	Текущий элемент ПОЛИЗа	Операция	Таблицы переменных	
			адреса	значения
пуст	15	метка – в стек	1) a 2) b	1) 7 2) 10
19	16	переход к элементу ПОЛИЗ с номером, извлекаемым из верхушки стека	1) a 2) b	1) 7 2) 10
пуст	19	интерпретация завершена	1) a 2) b	1) 7 2) 10

Пример 2.19. Построим интерпретатор ПОЛИЗа для языка *M*.

Введем следующие обозначения процедур и функций:

1) *addr*(*l*) – функция, выдающая адрес ячейки, отведенной для хранения лексемы *l*;

2) *cont*(*A*) – функция, выдающая содержимое ячейки с адресом *A*;

3) *let*(*A*, *x*) – процедура записи в ячейку с адресом *A* значения *x*;

4) *inst*(*x*) – процедура записи в стек значения *x*;

5) *outst*(*x*) – процедура считывания из стека значения *x*.

Тело интерпретатора ПОЛИЗа будет иметь следующий вид:

```
 free:=1; {на начало P}
repeat
   LEX:=P[free]; {очередная лексема}
   n:=LEX[1]; k:=LEX[2];
   case n of
      0: inst(k); {метка - в стек}
      5: inst(addr(LEX)); {адрес - в стек}
      1,3,4: inst(cont(addr(LEX))); {значение - в стек}
      2: {знак операции}
         case k of
            8{+}: begin outst(y); outst(x); inst(x+y) end;
            9{-}: begin outst(y); outst(x); inst(x-y) end;
            …{аналогично для *, / и других операций}
            14{¬}: begin outst(x); inst(not x) end;
            5{:=} begin outst(x); outst(A); let(A, x) end;
            18{!}: begin outst(free); free:=free-1 end;
            19{!F}: begin
                        outst(free1); outst(B);
                        if B=false then free:=free1-1;
                    end;
            20{R}: begin outst(A); read(x); let(A, x) end;
            21{W}: begin outst(x); write(x) end
         end
```

```
        end
        free:=free+1;
    until (k=2) and (n=2);
```

3 Постановка задачи к курсовой работе

Разработать компилятор модельного языка, 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 учебного 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) Записать правила вывода грамматики с действиями по переводу в ПОЛИЗ программы на модельном языке.

12) Пополнить разработанное программное средство процедурами, реализующими генерацию внутреннего представления введенной программы в форме ПОЛИЗа.

13) Разработать интерпретатор ПОЛИЗа программы на модельном языке.

14) Составить набор контрольных примеров, демонстрирующих:

а) все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке;

б) перевод в ПОЛИЗ различных конструкций языка;

в) представить ход интерпретации синтаксически и семантически правильной программы с помощью таблицы.

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 анализ
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 компиляторов, 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 компилятора 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 структуру, 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 ПЭВМ. 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 СТП101-00.

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

5 Индивидуальные варианты задания

ПРИМЕЧАНИЕ: В вариантах заданий конструкций модельного языка следует отличать фигурные и круглые скобки, являющиеся символами модельного языка (они заключены в кавычки «», например «(») от фигурных и круглых скобок - элементов металингвистических формул.

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*

4	<операции_группы_отношения>::=*NEQ |EQV |LOWT |LOWE |GRT |GRE*

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*
4	<операции_группы_сложения>:: = *add* | *disa* | ||

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*
4	<операции_группы_умножения>::= *umn* | *del* | &&

Таблица 5.4 - Унарная операция

Номер	Синтаксис операции
1	<унарная_операция>::= *not*
2	<унарная_операция>::= !
3	<унарная_операция>::= ~
4	<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*
4	<программа>::= begin(<описание> | <оператор>) {; (<описание> | <оператор>)} *end*
5	<программа>::= *begin var* <описание> {; <оператор>} *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> }
4	<описание>::= var{<идентификатор> {, <идентификатор> } : <тип> ;}
5	<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*
4	<тип>::=# | @ | &
5	<тип>::= % | *real* | $

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> } «}»
4	<составной>::=«[» <оператор> { (: | перевод строки) <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>
4	<присваивания>::= <идентификатор> *assign* <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*
4	<условный>::= *if* <выражение> *then* <оператор> [*else* <оператор>] *end*

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>
4	<фиксированного_цикла>::= *for* <присваивания> *val* <выражение> *do* <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*
4	<условного_цикла>::= *while* <выражение> *do* <оператор>*next*

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>}«)»
4	<ввода>::= *enter* <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> }«)»
4	<вывода>::= *displ* <выражение> {, <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	(*	*)
4	%	%
5	#	#

Таблица 5.16 – Индивидуальные номера вариантов

Номер варианта	Номер задания	Номер варианта	Номер здания
1	111111	69	141222
2	122211	70	224314
3	113211	71	214314
4	113311	72	344311
5	121132	73	142132
6	121212	74	142133
7	123112	75	224234
8	123312	76	214334
9	131111	77	332423
10	132111	78	131315
11	211121	79	313434
12	213222	80	121215
13	213321	81	323431
14	221122	82	122325
15	222222	83	333413
16	223122	84	111235
17	223322	85	322421
18	231123	86	112315
19	232223	87	313423
20	233323	88	121235
21	311111	89	132225
22	311211	90	331412
23	311311	91	113135
24	332211	92	321411
25	313311	93	123135
26	321122	94	322422
27	321222	95	133115
28	323122	96	311434
29	331133	97	122325
30	331233	98	312412
31	311113	99	113125
32	321133	100	321434

33	332222	101	455433
34	312131	102	415344
35	321132	103	143422
36	313213	104	251145
37	311111	105	332442
38	333333	106	421415
39	323232	107	135342
40	313212	108	453435
41	322133	109	155423
42	321322	110	455344
43	231231	111	453141
44	312312	112	341435
45	332323	113	452312
46	213112	114	325244
47	132231	115	451412
48	213333	116	344341
49	232322	117	255422
50	322313	118	253443
51	131312	119	155445
52	122311	120	125411
53	243314	121	455214
54	342334	122	335443
55	241324	123	453411
56	234324	124	355432
57	344314	125	425421
58	341111	126	252443
59	234234	127	255135
60	312324	128	445441
61	143314	129	115432
62	243334	130	453443
63	344324	131	113431
64	344133	132	411345
65	244123	133	453434
66	142334	134	155411
67	241314	135	444415
68	244132	136	354445

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) Изобразите графические примитивы диаграмм Вирта.

5) Дайте определение понятию «компилятор».

6) Каждый ли компилятор является транслятором?

7) Назовите известные Вам компилируемые языки программирования.

8) Перечислите основные функции компилятора.

9) Назовите этапы компиляции.

10) Охарактеризуйте общую схему работы компилятора.

11) Что называется проходом компилятора?

12) Что называется лексемой языка программирования?

13) Какие задачи выполняет лексический анализатор программы?

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

15) Перечислите основные группы лексем языков программирования.

16) Что представляет собой диаграмма состояний с действиями?

17) Расскажите алгоритм разбора цепочек по ДС с действиями.

18) Составьте диаграмму состояний с действиями для модельного языка.

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

20) Каково назначение синтаксического анализатора программы?

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

22) В чем сущность метода рекурсивного спуска?

23) Назовите необходимые условия применимости метода рекурсивного спуска.

24) Какие эквивалентные преобразования КС-грамматик Вам известны?

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

26) Какой вывод цепочки грамматики называется левосторонним?

27) В чем заключается специфика синтаксически управляемого перевода?

28) Перечислите основные задачи семантического анализатора.

29) Предложите один из возможных способов обработки описаний программы.

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

31) Дайте сравнительную характеристику известных форм внутреннего представления программы.

32) Запишите правила перевода в ПОЛИЗ выражений и операторов модельного языка.

Список использованных источников

1 Афанасьев А.Н. Формальные языки и грамматики: Учебное пособие. – Ульяновск: УлГТУ, 1997. – 84с.

2 Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Изд. дом «Вильямс», 2001. – 768с.

3 Братчиков И.Л. Синтаксис языков программирования / Под ред. С.С. Лаврова. – М.: Наука, 1975. - 262с.

4 Вайнгартен Ф. Трансляция языков программирования / Под ред. Мартынюка В.В.- М.: Мир, 1977. - 192с.

5 Вильямс А. Системное программирование в Windows 2000 для профессионалов. – СПб.: Питер, 2001. – 624с.

6 Волкова И.А., Руденко Т.В. Формальные языки и грамматики. Элементы теории трансляции. – М.: Диалог-МГУ, 1999. – 62с.

7 Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб: Питер, 2001. – 736с.

8 Грис Д. Конструирование компиляторов для цифровых вычислительных машин: Пер. с англ. – М.: Мир, 1975. – 544с.

9 Дворянкин А.И. Основы трансляции: Учебное пособие. – Волгоград: ВолгГТУ, 1999. – 80с.

10 Жаков В.И., Коровинский В.В., Фильчаков В.В. Синтаксический анализ и генерация кода. – СПб.: ГААП, 1993. – 26с.

11 Ишакова Е.Н. Теория формальных языков, грамматик и автоматов: Методические указания к лабораторному практикуму. – Оренбург: ГОУ ВПО ОГУ, 2004. – 54с.

12 Компаниец Р.И., Маньков Е.В., Филатов Н.Е. Системное программирование. Основы построения трансляторов. – СПб.: Корона принт, 2000. – 256с.

13 Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979. - 654с.

14 Пантелеева И.А. Методы трансляции: Конспект лекций. – Новосибирск: Изд-во НГТУ, 1998. – Ч.2. – 51с.

15 Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под ред. А. Матросова. – СПб: Питер, 2002. – 688с.

16 Рейуорд-Смит В. Теория формальных языков. Вводный курс: Пер. с англ. – М.: Радио и связь, 1988. – 128с.

17 Серебряков В.И. Лекции по конструированию компиляторов. – М.: МГУ, 1997. – 171с.

18 Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учеб. пособие. – М.: Финансы и статистика, 2004. – 320с.

19 Федоров В.В. Основы построения трансляторов: Учебное пособие. – Обнинск: ИАТЭ, 1995. – 105с.

20 Хантер Р. Проектирование и конструирование компиляторов: Пер. с англ. – М.: Финансы и статистика, 1984. – 232с.

Приложение А

(обязательное)

Алгоритм решения задачи

Укрупненная схема алгоритма программного средства представлена на рисунке 6.1.

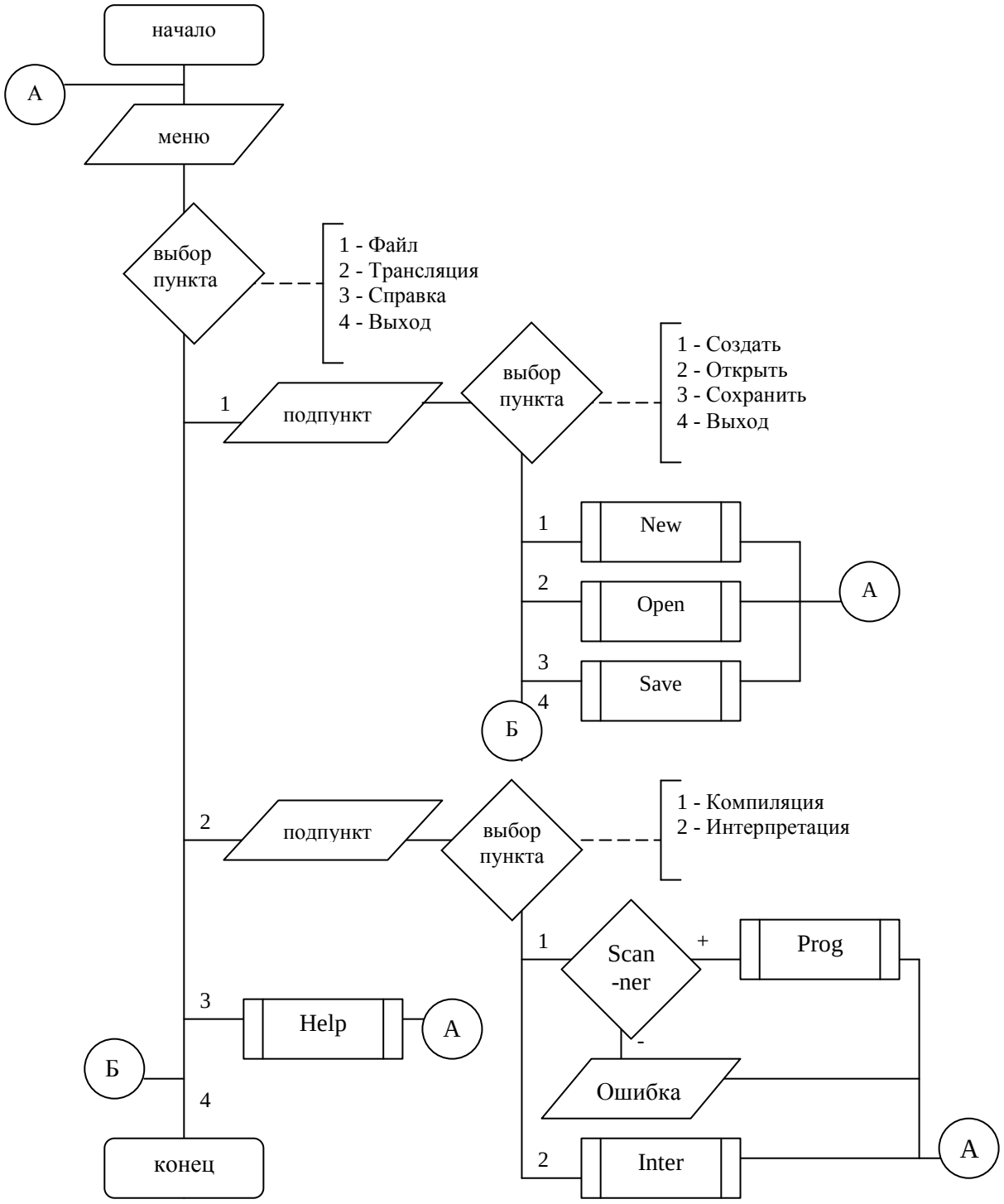


Рисунок 6.1 – Укрупненная схема алгоритма программного средства

Приложение В
(обязательное)
Контрольный пример

Результаты работы лексического анализатора представлены на рисунке А.1.

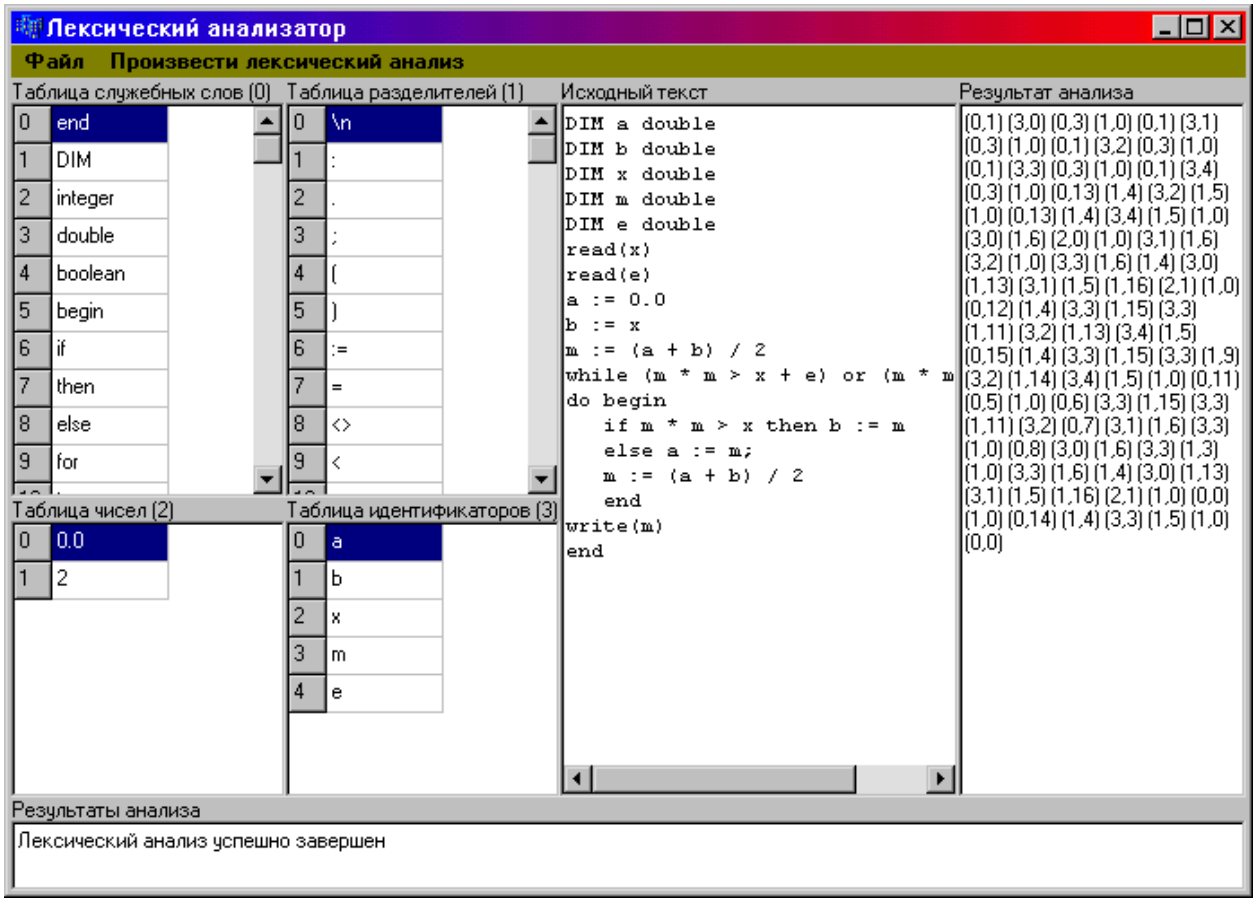


Рисунок А.1 – Выходные данные лексического анализатора

Приложение С
(обязательное)
Текст программы

la.h

```
#include <grids.hpp>
#include <fstream.h>
#include <string.h>
#include <vector>
#include <string>

using std::string;
using std::vector;

// структура, описывающая лексему
struct par{
   long n; // номер таблицы
   long k; // номер в таблице
};

typedef vector<string> wordtable;
typedef vector<par> parvec;
// состояния диаграммы
enum states {SH, // начало
                     SI, // идентификатор
                     SN, // число (до точки)
                     SND, // дробная часть
                     SNS, // знак порядка
                     SNP, // порядок
                     SO, // ограничитель
                     SC, // комментарий
                     SL, // <
                     SG, // >
                     SS, // :
                     SDT, // .
                     SER, // ошибка
                     SV}; // выход

class LA;

// класс сканер
class Scanner{
   public:
      LA * A; // связанный лексический анализатор
      string instr; // входная строка с исходным текстом
     unsigned long pos; // позиция в строке
      long z; // найденная позиция в таблице
      long errcode; // код ошибки
      char cur; // текущий символ
      string S; // строка, формирующая лексему
      states State; // состояние дмаграммы
int Scan(); // метод-сканер
      char gc(){ // считывание следующего символа
         if (pos >= instr.size()){
            State = SV;
            return cur;
         }
         return (cur = instr[pos++]);
      }
      bool letter(){ // проверка символа на букву
         return isalpha(cur);
      }
      bool digit(){ // проверка символа на цифру
         return isdigit(cur);
      }
      long look(wordtable * t); // поиск лексемы S в таблице t
      long put(wordtable * t){ // помещение лексемы в таблицу
         z = look(t);
         if (z >= 0)
            return z;
         t->push_back(S);
         return (z = (t->size() - 1));
      }
      void out(long n, long z);
};

// класс лексический анализатор
class LA{
   public:
      wordtable R; // таблица служебных слов
      wordtable D; // таблица разделителей
      wordtable I; // таблица идентификаторов
      wordtable N; // таблица чисел
      parvec res; // вектор пар чисел - результат лексического анализа

  Scanner S; // сканер
      void InTables(char *fname); // ввод таблиц R и D из файла
      void OutTable(TStringGrid * G, wordtable * X); // вывод таблицы в StringGrid
      int Scan(string s); // сканирование
      string ErrorMsg(int code); // сообщение об ошибке по коду
      string GetResult(); // сформировать результат в виде строки
            void OutTables(char *fname); // вывод таблиц I и N в файл
            void OutResult(char *fname); // вывод результата в файл
         };
```

la.cpp

```
#include "la.h"

// перевод строки в нижний регистр
void tolower(string &s){
   for (unsigned long i = 0; i < s.length(); i++)
      s[i] = tolower(s[i]);
}

int Scanner::Scan(){
   par t;
   pos = 0;
   State = SH;
   errcode = 0;
   gc();
   while(State != SV && State != SER){
      while(State != SV && cur != '\n' &&
isspace(cur))
         gc();
      if (State == SV)
         break;
      if (letter()){ // буква
         State = SI;
         S = cur;
         for(gc(); State != SV && (letter() || dig-
it()); gc())
            S += cur;
         //tolower(S);
         look(&A->R);
         if (z >= 0)
            out(0, z);
         else {
            put(&A->I);
            out(3, z);
         }
      } else if (digit()){ //число
         State = SN;
         S = cur;
         for(gc(); State != SV && (digit() ||
strchr("ABCDEFabcdef", cur)); gc())
            S += cur;
         if (strchr("HhDdOoBb", cur)){
            S += cur;
            gc();
         } else if (cur == '.'){ // дробная часть
          State = SND;
            S += cur;
            for(gc(); State != SV && digit();
gc())
               S += cur;
            if (cur == 'e' || cur == 'E'){ // порядок
                  S += cur;
               gc();
               State = SNS;
               if (cur == '+' || cur == '-'){
                  S += cur;
                  gc();
               }
               State = SNP;
               for(; State != SV && digit(); gc())
                  S += cur;
            }
         } else if ((digit() || cur == '+' || cur == '-')
&& (S[S.length() - 1] == 'e' || S[S.length() - 1] ==
'E')){ // порядок
               State = SNP;
            for(gc(); State != SV && digit(); gc())
               S += cur;
         }
         put(&A->N);
         out(2, z);
      } else if (cur == '{'){ // комментарий
         State = SC;
         for(gc(); State != SV && cur != '}'; gc());
         if (State == SV){
            errcode = 1;
            break;
         }
         gc();
      } else if (cur == '<'){ // < <= <>
         State = SL;
         gc();
         if (cur == '=' || cur == '>'){
            S = "<";
            S += cur;
            gc();
         }
         else
            S = "<";
         look(&A->D);
         out(1, z);
      } else if (cur == '>'){ // > >=
         State = SG;
         gc();
         if (cur == '='){
            S = ">=";
            gc();
         }
         else
            S = ">";
         look(&A->D);
         out(1, z);
      } else if (cur == ':'){ // : :=
         State = SS;
         gc();
         if (cur == '='){
            S = ":=";
            gc();
         }
         else
            S = ":";
         look(&A->D);
         out(1, z);
```