## Практическая работа №4

*Задание:* написать на любом языке программирования (или доработать листинг 1) лексический анализатор на базе конечного автомата входного языка, описанного диаграммой состояний рис. 1.

Входной язык, содержит операторы цикла **for (…; …; …) do …**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

Описанный выше входной язык может быть задан с помощью КС-грамматики **G** ({**for**, **do**, ‘:=’, ‘<’, ‘>’, ‘=’, ‘-’, ‘+’, ‘**(**‘, ‘**)**’, ‘;’, ‘.’, ‘_’, ‘a’, ‘b’, ‘c’, ..., ‘x’, ‘y’, ‘z’, ‘A’, ‘B’, ‘C’, ..., ‘X’, ‘Y’, ‘Z’, ‘0’, ‘1’, ‘2’, ’3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’}, {*S, A, B, C, I, L, N, Z*}, **P**, *S*) с правилами **P** (правила представлены в расширенной форме Бэкуса-Наура):

$S \rightarrow$ **for (***A*; *A*; *A*;**) do** *A*; | **for (***A*; *A*; *A*;**) do** *S*; | **for (***A*; *A*; *A*;**) do** *A*; S

$A \rightarrow I := B$

$B \rightarrow C > C \mid C < C \mid C = C$

$C \rightarrow I \mid N$

$I \rightarrow (\_ \mid L)\ \{\_ \mid L \mid Z \mid 0\}$

$N \rightarrow$ [- | +] ({0 | *Z*} . {0 | *Z*} | {0 | *Z*}. | {0 | *Z*}) [(e | E) [- | +]{0 | *Z*}][f | l | F | L]

$L \rightarrow a \mid b \mid c \mid \ldots \mid x \mid y \mid z \mid A \mid B \mid C \mid \ldots \mid$ X | Y | Z

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

Целевым символом грамматики является символ *S.* Лексемы входного языка разделим на несколько классов:

- ключевые слова языка (**for**, **do**) – класс 1;
- разделители и знаки операций (‘**(**‘, ‘**)**’, ‘;’, '<', ‘>’, ‘=’) – класс 2;
- знак операции присваивания (‘:=’) – класс 3;
- идентификаторы – класс 4;
- десятичные числа с плавающей точкой (в обычной и экспоненциальной форме) – класс 5.

Границами лексем будут служить пробелы, знаки табуляции, знаки перевода строки и возврата каретки, круглые скобки, точка с запятой и знак двоеточия. При этом круглые скобки и точка с запятой сами являются лексемами, а знак двоеточия, являясь границей лексемы, в то же время является и началом другой лексемы – операции присваивания.

Диаграмма состояний для лексического анализатора приведена на рис. 1. Состояния на диаграмме соответствуют классам лексем (см. таблицу 1). А действия – вызовом функций в программе, реализующей лексический анализатор.

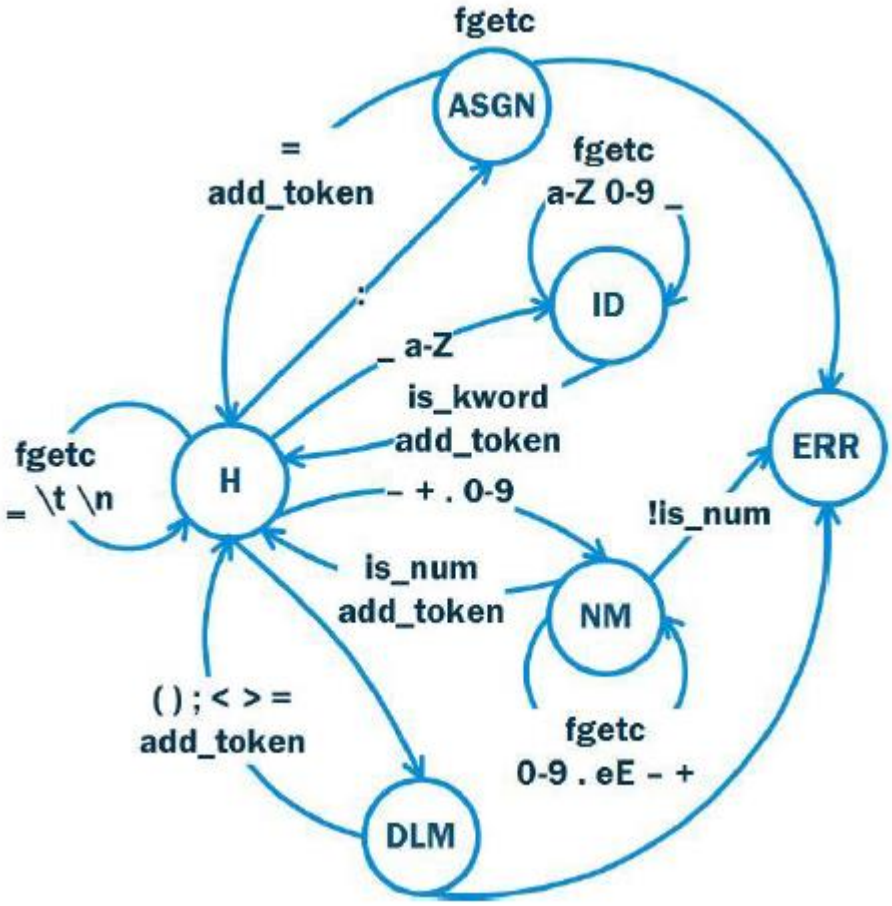


Рис. 1. Диаграмма состояний лексического анализатора

Таблица 1. Состояния и действия для диаграммы

| **Состояния** | |
|---|---|
| H | Начальное состояние |
| ID | Идентификаторы |
| NM | Числа |
| ASGN | Знак присваивания (:=) |
| DLM | Разделители (;, (, ), =, >, <) |
| ERR | Нераспознанные символы |
| **Действия** | |
| fgetc | чтение символа из файла |
| is_kword | проверка, является ли идентификатор ключевым словом |
| is_num | проверка на правильность записи числа |
| add_token | добавление токена в таблицу лексем |

В листинге 1 приведен пример программной реализации лексического анализатора. Функция `lexer` реализует алгоритм, описываемый конечным автоматом (рис. 1). Переменная `CS` содержит значение текущего состояния автомата. В начале работы программы – это начальное состояние `H`. Переход из этого состояния в другие происходит только, если во входной последовательности встречается символ, отличный от пробела, знака табуляции или перехода на новую строку. После достижения границы лексемы осуществляется возврат в начальное состояние. Из состояния `ERR` тоже

происходит возвращение в начальное состояние, таким образом, лексический анализ не останавливает после обнаружения первой ошибки, а продолжается до конца входной последовательности. Концом входной последовательности является конец файла.

Листинг 1:

```
#define NUM_OF_KWORDS 2
char *keywords[NUM_OF_KWORDS] = {"for", "do"};
enum states {H, ID, NM, ASGN, DLM, ERR};
enum tok_names {KWORD, IDENT, NUM, OPER, DELIM};
struct token
{
      enum tok_names token_name;
      char *token_value;
};
struct lexeme_table
{
      struct token tok;
      struct lexeme_table *next;
};
struct lexeme_table *lt = NULL;
struct lexeme_table *lt_head = NULL;
int lexer(char *filename);
int is_kword(char *id);
int add_token(struct token *tok);
int lexer(char *filename)
{
      FILE *fd;
      int c, err_symbol;
      struct token tok;
      if((fd = fopen(filename, "r")) == NULL)
      {
            printf("\nCannot open file %s.\n", filename);
            return -1;
      }
      enum states CS = H;

      c = fgetc(fd);

      while(!feof(fd))
      {
            switch(CS)
            {
                  case H:
                  {
                        while((c == ' ') || (c == '\t') || (c == '\n'))
                        {
                              c = fgetc(fd);
                        }
                        if(((c >= 'A') && (c <= 'Z')) ||
                        ((c >= 'a') && (c <= 'z')) || (c == '_'))
                        {
                              CS = ID;
```

```
      }else if(((c >= '0') && (c <= '9')) || (c == '.')
      ||
      (c == '+') || (c == '-'))
      {
            CS = NM;
      }else if(c == ':')
      {
            CS = ASGN;
      }else{
            CS = DLM;
      }
      break;
      }// case H

case ASGN:
{
      int colon = c;
      c = fgetc(fd);
      if(c == '=')
      {
            tok.token_name = OPER;
            if((tok.token_value                    =(char
            *)malloc(sizeof(2))) == NULL)
            {
                  printf("\nMemory  allocation  error  in
                  function \"lexer\"\n");
                  return -1;
            }
            strcpy(tok.token_value, ":=");
            add_token(&tok);
            c = fgetc(fd);
            CS = H;
      }else{
      err_symbol = colon;
      CS = ERR;
      }
      break;
}// case ASGN

case DLM:
{
      if((c == '(') || (c == ')') || (c == ';'))
      {
            tok.token_name = DELIM;
            if((tok.token_value =
            (char *)malloc(sizeof(1))) == NULL)
            {
                  printf("\nMemory  allocation  error  in
                  function
                  \"lexer\"\n");
                  return -1;
            }
            sprintf(tok.token_value, "%c", c);
            add_token(&tok);
            c = fgetc(fd);
            CS = H;
      }else if((c == '<') || (c == '>') || (c == '='))
      {
```

```
                tok.token_name = OPER;
                if((tok.token_value =
                (char *)malloc(sizeof(1))) == NULL)
                {
                     printf("\nMemory allocation error in
                     function
                     \"lexer\"\n");
                     return -1;
                }
                sprintf(tok.token_value, "%c", c);
                add_token(&tok);
                c = fgetc(fd);
                CS = H;
           }else{
                err_symbol = c;
                c = fgetc(fd);
                CS = ERR;
           }// if((c == '(') || (c == ')') || (c == ';'))
           break;
}// case DLM

case ERR:
{
printf("\nUnknown character: %c\n", err_symbol);
CS = H;
break;
}
case ID:
{
      int size = 0;
      char buf[256];
      buf[size] = c;
      size++;
      c = fgetc(fd);
      while(((c >= 'A') && (c <= 'Z')) || ((c >= 'a') &&
      (c <= 'z')) || ((c >= '0') && (c <= '9')) ||
      (c == '_'))
      {
           buf[size] = c;
           size++;
           c = fgetc(fd);
      }
      buf[size] = '\0';
      if(is_kword(buf))
      {
           tok.token_name = KWORD;
      }else{
           tok.token_name = IDENT;
      }
      if((tok.token_value = (char *)malloc(strlen(buf)))
      == NULL)
      {
           printf("\nMemory allocation error in function
           \"lexer\"\n");
           return -1;
      }
      strcpy(tok.token_value, buf);
      add_token(&tok);
```

```
                    CS = H;
                    break;
                } // case ID
                .
                .
                .
            } // switch
        } // while
} // int lexer(…)
```