Лабораторная работа №2. Лексический анализатор

Теоретические сведения

Первая фаза компиляции называется **лексическим анализом** или сканированием.

> **Лексический анализатор (сканер)** читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы сканер строит выходной *токен* (англ. token – знак, символ) вида

‹*имя_токена, значение_атрибута*›

Первый компонент токена, *имя_токена*, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Предположим, например, что исходная программа содержит инструкцию присваивания

```
a = b + c * d
```

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены:

1) `a` представляет собой лексему, которая может отображаться в токен ‹*id*, 1›, где *id* – абстрактный символ, обозначающий идентификатор, а 1 указывает запись в таблице идентификаторов для `a`, в которой хранится такая информация как имя и тип идентификатора;
2) символ присваивания = представляет собой лексему, которая отображается в токен ‹=›. Поскольку этот токен не требует значения атрибута, второй компонент данного токена

опущен. В качестве имени токена может быть использован любой абстрактный символ, например, такой, как «assign», но для удобства записи в качестве имени абстрактного символа можно использовать саму лексему;

3) `b` представляет собой лексему, которая отображается в токен ‹*id*, 2›, где 2 указывает на запись в таблице идентификаторов для `b`;
4) `+` является лексемой, отображаемой в токен ‹+›;
5) `c` – лексема, отображаемая в токен ‹*id*, 3›, где 3 указывает на запись в таблице идентификаторов для `c`;
6) `*` – лексема, отображаемая в токен ‹*›;
7) `d` – лексема, отображаемая в токен ‹*id*, 4›, где 4 указывает на запись в таблице идентификаторов для `d`.

Пробелы, разделяющие лексемы, лексическим анализатором пропускаются.

Представление инструкции присваивания после лексического анализа в виде последовательности токенов примет следующий вид:

‹*id*, 1›‹=›‹*id*, 2›‹+›‹*›‹*id*, 3›‹*id*, 4›.

В таблице 1 приведены некоторые типичные токены, неформальное описание их шаблонов и некоторые примеры лексем. *Шаблон* (англ. pattern) – это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой просто последовательность символов, образующих это ключевое слово.

Чтобы увидеть использование этих концепций на практике, рассмотрим инструкцию на языке программирования Си

```
printf("Total = %d\n", score);
```

в которой `printf` и `score` представляют собой лексемы, соответствующие токену *id*, а `"Total = %d\n"` является лексемой, соответствующей токену *literal*.

Таблица 1. Примеры токенов

Токен	Неформальное описание	Примеры лексем
if	Символы i, f	`if`
else	Символы e, l, s, e	`else`
comp	< или > или <= или >= или == или !=	`<=`
id	Буква, за которой следуют буквы и цифры	`score, D2`
number	Любая числовая константа	`3.14159`
literal	Последовательность любых символов, заключённая в кавычки (кроме самих кавычек)	`“Total = %d\n”`

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ:

– упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объём обрабатываемой информации, так как лексический анализатор

структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию;

- для выделения в тексте и разбора лексем можно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
- сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

В большинстве компиляторов лексический и синтаксический анализаторы – это взаимосвязанные части. Лексический разбор исходного текста в таком варианте выполняется поэтапно, так что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к сканеру за следующей лексемой. При этом он может сообщить информацию о том, какую лексему следует ожидать. В процессе разбора может даже происходить «откат назад», чтобы выполнить анализ текста на другой основе. В дальнейшем будем исходить из предположения, что все лексемы могут быть однозначно выделены сканером на этапе лексического разбора.

Работу синтаксического и лексического анализаторов можно изобразить в виде схемы, приведённой на рисунке 1.

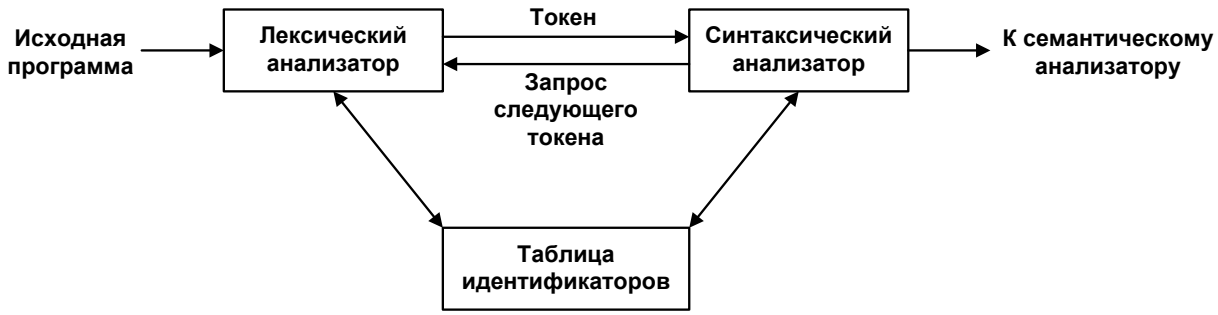

Рисунок 1. Взаимодействие лексического анализатора с синтаксическим

В простейшем случае фазы лексического и синтаксического анализа могут выполняться компилятором последовательно. Но для многих языков программирования информации на этапе лексического анализа может быть недостаточно для однозначного определения типа и границ очередной лексемы.

Иллюстрацией такого случая может служить пример оператора присваивания из языка программирования Си

```
k = i+++++j;
```

который имеет только одну верную интерпретацию (если операции разделить пробелами):

```
k = i++ + ++j;
```

Если невозможно определить границы лексем, то лексический анализ исходного текста должен выполняться поэтапно. Тогда лексический и синтаксический анализаторы должны функционировать параллельно. Лексический анализатор, найдя очередную лексему, передаёт её синтаксическому анализатору, тот пытается выполнить анализ считанной части исходной программы и может либо запросить у лексического анализатора следующую лексему, либо потребовать от него вернуться на несколько шагов назад и попробовать выделить лексемы с другими границами.

Очевидно, что параллельная работа лексического и синтаксического анализаторов более сложна в реализации, чем их последовательное выполнение. Кроме того, такая реализация требует больше вычислительных ресурсов и в общем случае большего времени на анализ исходной программы.

Чтобы избежать параллельной работы лексического и синтаксического анализаторов, разработчики компиляторов и языков программирования часто идут на разумные ограничения синтаксиса входного языка. Например, для языка Си принято соглашение, что при возникновении проблем с определением границ лексемы всегда выбирается лексема максимально возможной длины. Для рассмотренного выше оператора присваивания это приводит к тому, что при чтении четвёртого знака + из двух вариантов лексем (+ – знак сложения, ++ – оператор инкремента) лексический анализатор выбирает самую длинную, т.е. ++, и в целом весь оператор будет разобран как

```
k = i++ ++ +j;
```

что неверно. Компилятор gcc в этом случае выдаст сообщение об ошибке: lvalue required as increment operand – в качестве операнда оператора инкремента требуется l-значение. Любые неоднозначности при анализе данного оператора присваивания могут быть исключены только в случае правильной расстановки пробелов в исходной программе.

Вид представления информации после выполнения лексического анализа целиком зависит от конструкции компилятора. Но в общем виде её можно представить как **таблицу лексем**, которая в каждой строчке должна содержать информацию о виде лексемы, её типе и, возможно, значении. Обычно такая таблица имеет два столбца: первый – строка лексемы, второй – указатель на информацию о лексеме, может быть включён и третий столбец – тип лексем. **Не следует путать таблицу лексем и таблицу идентификаторов – это две принципиально разные таблицы!** Таблица лексем содержит весь текст исходной программы, обработанный лексическим анализатором. В неё входят все возможные типы лексем, при этом, любая лексема может в ней встречаться любое число раз. Таблица идентификаторов содержит только следующие типы лексем: идентификаторы и константы. В неё не попадают ключевые слова входного языка, знаки операций и разделители. Каждая лексема в таблице идентификаторов может встречаться только один раз.

Вот пример фрагмента текста программы на языке Паскаль и соответствующей ему таблицы лексем:

```
...
begin
 for i:=1 to N do
  fg := fg * 0.5
...
```

Таблица 2. Таблица лексем программы

Лексема	Тип лексемы	Значение
`begin`	Ключевое слово	X1
`for`	Ключевое слово	X2
`i`	Идентификатор	i : 1
`:=`	Знак присваивания	
`1`	Целочисленная константа	1
`to`	Ключевое слово	X3
`N`	Идентификатор	N : 2
`do`	Ключевое слово	X4
`fg`	Идентификатор	fg : 3
`:=`	Знак присваивания	
`fg`	Идентификатор	fg : 3
`*`	Знак арифметической операции	
`0.5`	Вещественная константа	0.5

В таблице 2 поле «значение» подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем. Значения, приведённые в примере, являются условными. Конкретные коды выбираются разработчиками при реализации компилятора. Связь между таблицей лексем и таблицей идентификаторов отражена в примере некоторым индексом, следующим после идентификатора за знаком «:». В реальном компиляторе эта связь определяется его реализацией.

Разработка лексического анализатора

В качестве примера возьмем входной язык, содержащий 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 | C < C | C = C

$C \rightarrow I \mid N$

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

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

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

$Z \rightarrow$ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 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. 2. Состояния на диаграмме соответствуют классам лексем (см. таблицу 3). А действия – вызовом функций в программе, реализующей лексический анализатор.

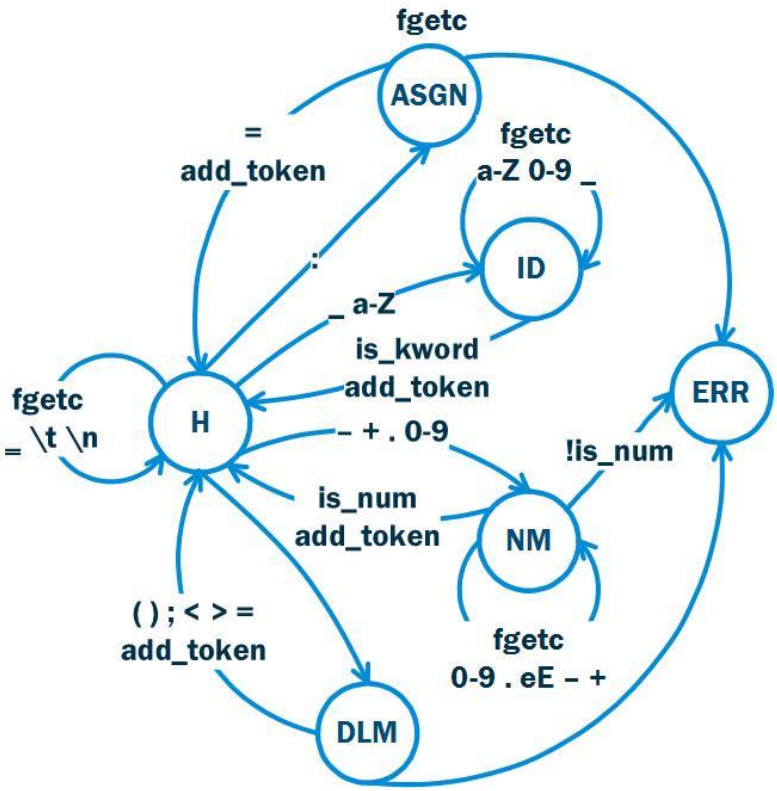

Рисунок 2. Диаграмма состояний лексического анализатора

Таблица 3. Состояния и действия для диаграммы, изображенной на рис. 2

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. 2). 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. Лексический анализатор

```
#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;
     }
```

Листинг 1. Лексический анализатор

```
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;
```

Листинг 1. Лексический анализатор

```
 }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);
```

Листинг 1. Лексический анализатор

```
            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(…)
```

Генератор лексических анализаторов Flex

Существуют различные программные средства для решения задачи построения лексических анализаторов. Наиболее известным из них является Lex (в более поздних версиях – Flex).

Программный инструментарий Flex позволяет определить лексический анализатор с помощью регулярных выражений для описания шаблонов токенов. Входные обозначения для Flex обычно называют *языком Flex*, а сам инструмент – *компилятором Flex*. Компилятор Flex преобразует входные шаблоны в конечный автомат и генерирует код (в файле с именем `lex.yy.c`), имитирующий данный автомат.

Рисунок 3. Схема использования Flex

На рисунке 3 показаны схема использования Flex и команды, соответствующие каждому этапу генерирования лексического анализатора. Входной файл `input.l` написан на языке Flex и описывает генерируемый лексический анализатор. Компилятор Flex преобразует `input.l` в программу на языке программирования Си (файл с именем `lex.yy.c`). При компиляции `lex.yy.c` необходимо прилинковать библиотеку Flex (`-lfl`). Этот файл компилируется в файл с именем a.out как обычно. Выход компилятора Си представляет собой работающий лексический анализатор, который на основе потока входных символов выдаёт поток токенов.

Обычно полученный лексический анализатор, используется в качестве подпрограммы синтаксического анализатора.

Структура программы на языке Flex имеет следующий вид:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

Обязательным является наличие правил трансляции, а, следовательно, и символов `%%` перед ними. Правила могут и отсутствовать в файле, но %% должны присутствовать всё равно.

Пример самого короткого файла на языке Flex:

```
%%
```

В этом случае входной поток просто посимвольно копируется в выходной. По умолчанию, входным является стандартный входной поток (stdin), а выходным – стандартный выходной (stdout).

Раздел объявлений может включать объявления переменных, именованные константы и регулярные определения (например, `digit [0-9]` – регулярное выражение, описывающее множество цифр от 0 до 9). Кроме того, в разделе объявлений может помещаться символьный блок, содержащий определения на Си. Символьный блок всегда начинается с `%{` и заканчивается `%}`. Весь код символьного блока полностью копируется в начало генерируемого файла исходного кода лексического анализатора.

Второй раздел содержит правила трансляции вида

```
Шаблон { Действие }
```

Каждый шаблон является регулярным выражением, которое может использовать регулярные определения из раздела объявлений. Действия представляют собой фрагменты кода, обычно написанные на языке программирования Си, хотя существуют и разновидности Flex для других языков программирования.

Третий раздел содержит различные дополнительные функции на Си, используемые в действиях. Flex копирует эту часть кода в конец генерируемого файла.

Листинг 2. Пример программы для подсчёта символов, слов и строк во введённом тексте

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.         { chars++; }
%%
int main(int argc, char **argv)
{
   yylex();
   printf("%8d%8d%8d\n", lines, words, chars);
   return 0;
}
```

В листинге 2 определены все три раздела программы на Flex.

В первом разделе объявлены три переменных-счётчика для символов, слов и строк, соответственно. Эта часть кода будет полностью скопирована в файл `lex.yy.c`.

Во втором разделе определены шаблоны токенов и действия, которые нужно выполнить при соответствии входного потока тому либо иному шаблону. **Перед шаблоном не должно быть пробелов, табуляций и т.п., поскольку Flex рассматривает любую строку, начинающуюся с пробела, как код, который нужно скопировать в файл `lex.yy.c`.**

В данном примере определены три шаблона:

1) `[a-zA-Z]+` соответствует слову текста. В соответствии с этим шаблоном слово может содержать прописные и заглавные буквы латинского алфавита. А знак + означает, что слово может состоять из одного или нескольких символов, описанных перед +. В случае

совпадения входной последовательности и этого шаблона, увеличиваются счётчики для слов и символов. Массив символов `yytext` всегда содержит текст, соответствующий данному шаблону. В нашем случае он используется для расчёта длины слова;

2) `\n` соответствует символу перевода строки. В случае совпадения входного потока с данным шаблоном происходит увеличение счётчиков для символов и строк на 1;
3) `.` является шаблоном для любого входного символа.

В функции `main` вызывается `yylex()` – функция, непосредственно выполняющая лексический анализ входного текста.

Ниже приведены команды для компиляции и запуска программы на языке Flex для подсчёта символов, слов и строк в тексте, введённом с клавиатуры.

```
$ flex words.l
$ gcc lex.yy.c -lfl
$ ./a.out
To be, or not to be: that is the question
       1      10      42
```

В таблице 4 перечислены специальные символы, использующиеся в регулярных выражениях (шаблонах) Flex.

Таблица 4. Специальные символы, использующиеся в регулярных выражениях Flex

Символ шаблона	Значение
`.`	Соответствует любому символу, кроме `\n`
`[]`	Класс символов, соответствующий любому из символов, описанных внутри скобок. Знак `'-'` указывает на диапазон символов. Например, `[0-9]` означает то же самое, что и `[0123456789]`, `[a-z]` – любая прописная буква латинского алфавита, `[A-z]` – все заглавные и прописные буквы латинского алфавита, а также 6 знаков пунктуации, находящихся между `Z` и `a` в таблице ASCII. Если символ `'-'` или `']'` указан в качестве первого символа после открывающейся квадратной скобки, значит он включается в описываемый класс символов. Управляющие (escape) последовательности языка Си также могут указываться внутри квадратных скобок, например, `\t`.
`^`	Внутри квадратных скобок используется как отрицание, например, регулярное выражение `[^\t\n]` соответствует любой последовательности символов, не содержащей табуляций и переводов строки. Если просто используется в начале шаблона, то означает начало строки.
`$`	При использовании в конце регулярного выражения означает конец строки.
`{}`	Если в фигурных скобках указаны два числа, то они интерпретируются как минимальное и максимальное количество повторений шаблона, предшествующего скобкам. Например, `A{1,3}`

Таблица 4. Специальные символы, использующиеся в регулярных выражениях Flex

Символ шаблона	Значение
	соответствует повторению буквы A от одного до трёх раз, а A{5} – 00000. Если внутри скобок находится имя регулярного определения, то это просто обращение к данному определению по его имени.
\	Используется в escape-последовательностях языка Си и для задания метасимволов, например, * – символ '*' в отличие от * (см. ниже).
*	Повторение регулярного выражения, указанного до *, 0 или более раз. Например, [\t]* соответствует регулярному выражению для пробелов и/или табуляций, отсутствующих или повторяющихся несколько раз.
+	Повторение регулярного выражения, указанного до +, один или более раз. Например, [0-9]+ соответствует строкам 1, 111 или 123456.
?	Соответствует повторению регулярного выражения, указанного до ?, 0 или 1 раз. Например, -?[0-9]+ соответствует знаковым числам с необязательным минусом перед числом.
|	Оператор «или». Например, true|false соответствует любой из двух строк.
()	Используются для группировки нескольких регулярных выражений в одно. Например, a(bc|de) соответствует входным последовательностям: abc или ade.
/	Так называемый присоединенный контекст. Например, регулярное выражение 0/1 соответствует 0 во входной строке 01, но не соответствует ничему в строках 0 или 02.
" "	Любое символы в кавычках рассматриваются как строка символов. Метасимволы, такие как *, теряют своё значение и интерпретируются как два символа: \ и *.

Лексический анализатор на языке Flex для грамматики из предыдущего раздела представлен в листинге 3.

В первой строке данной программы указаны опции, которые должны быть учтены при построении лексического анализатора. Для этого используется формат

```
%option имя_опции
```

Те же самые опции можно было бы указать при компиляции в командной строке как

```
--имя_опции
```

Для отключения опции перед её именем следует указать «no», как в случае с noyywrap. Полный список допустимых опций можно найти в документации по Flex [6, 8].

Первые версии генератора лексических анализаторов Lex вызывали функцию yywrap() при достижении конца входного потока yyin. В случае, если нужно было продолжить анализ входного текста из другого файла, yywrap возвращала 0 для продолжения сканирования. В противном случае возвращалась 1.

Листинг 3. Лексический анализатор на языке Flex

```
%option noyywrap yylineno
%{
    #include <stdio.h>
    int ch;
%}

digit[0-9]
letter[a-zA-Z]
delim[();]
oper[<>=]
ws[ \t\n]

%%

for { printf("KEYWORD (%d, %d): %s\n", yylineno, ch, yytext);
      ch += yyleng;
    }
do { printf("KEYWORD (%d, %d): %s\n", yylineno, ch, yytext);
     ch += yyleng;
   }
("_"|{letter})("_"|{letter}|{digit})* {
     printf("IDENTIFIER (%d, %d): %s\n", yylineno, ch, yytext);
     ch += yyleng;
  }
[-+]?({digit}*\.{digit}+|{digit}+\.|{digit}+)
                                        ([eE][-+]?{digit}+)?[flFL]? {
     printf("NUMBER (%d, %d): %s\n", yylineno, ch, yytext);
     ch += yyleng;
}
{oper} { printf("OPERATION (%d, %d): %s\n", yylineno, ch, yytext);
         ch += yyleng;
}
":=" { printf("OPERATION (%d, %d): %s\n", yylineno, ch, yytext);
       ch += yyleng;
     }
{delim} { printf("DELIMITER (%d, %d): %s\n", yylineno, ch, yytext);
          ch += yyleng;
        }
{ws}+ { ch += yyleng; }
. { printf("Unknown character (%d, %d): %s\n", yylineno, ch, yytext);
    ch += yyleng;
  }

%%

int main(int argc, char **argv)
{
     if(argc < 2)
```

Листинг 3. Лексический анализатор на языке Flex

```
    {
        printf("\nNot enough arguments. Please specify filename.\n");
        return -1;
    }
    if((yyin = fopen(argv[1], "r")) == NULL)
    {
        printf("\nCannot open file %s.\n", argv[1]);
        return -1;
    }
    ch = 1;
    yylineno = 1;
    yylex();
    fclose(yyin);
    return 0;
}
```

В современных версиях Flex рекомендуется отключать использование `yywrap` с помощью опции `noyywrap`, если в программе на языке Flex есть своя функция `main`, в которой и определяется, какой файл и когда сканировать.

Использование опции `yylineno` позволяет вести нумерацию строк входного файла и в случае ошибки сообщать пользователю номер строки, в которой эта ошибка произошла. Flex определяет переменную `yylineno` и автоматически увеличивает её значение на 1, когда встречается символ `'\n'`. При этом Flex не инициализирует эту переменную. Поэтому в функции `main` перед вызовом функции лексического анализа `yylex` переменной `yylineno` присваивается 1.

Flex, по умолчанию, присваивает переменной `yyin` указатель на стандартный поток ввода. Если предполагается сканировать текст из файла, то нужно присвоить переменной `yyin` результат вызова функции `fopen` до вызова `yylex`:

```
yyin = fopen(argv[1], "r");
```

В функции `main` в приведённом примере открывается файл, имя которого было указано пользователем при вызове лексического анализатора.

Входной текстовый файл `prog` содержит следующую программный код:

```
for(abc1:=.;abc1<11.0E+1;abc1:=abc1>.1)do abc1:=abc1=34E5;
```

Для компиляции и запуска программы используются следующие команды:

```
flex example.l
gcc lex.yy.c -o scanner -lfl
./scanner prog
```

Результат работы программы:

```
KEYWORD (1, 1): for
DELIMITER (1, 4): (
IDENTIFIER (1, 5): abc1
OPERATION (1, 9): :=
Unknown character (1, 11): .
DELIMITER (1, 12): ;
```

```
IDENTIFIER (1, 13): abc1
OPERATION (1, 17): <
NUMBER (1, 18): 11.0E+1
DELIMITER (1, 25): ;
IDENTIFIER (1, 26): abc1
OPERATION (1, 30): :=
IDENTIFIER (1, 32): abc1
OPERATION (1, 36): >
NUMBER (1, 37): .1
DELIMITER (1, 39): )
KEYWORD (1, 40): do
IDENTIFIER (1, 43): abc1
OPERATION (1, 47): :=
IDENTIFIER (1, 49): abc1
OPERATION (1, 53): =
NUMBER (1, 54): 34E5
DELIMITER (1, 58): ;
```

Символ «.» определяется лексическим анализатором как неизвестный, потому что до него и/или после него отсутствуют цифры. А значит это не вещественное с плавающей точкой, а просто «.». Такой символ не входит во множество терминальных символов языка, для которого создается лексический анализатор.

Задание на лабораторную работу

Для выполнения лабораторной работы необходимо:

1) написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов. Программа должна выдавать сообщения о наличии во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа;
2) в качестве вспомогательного средства для генерации кода лексического анализатора использовать Flex.

Варианты заданий

1. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и экспоненциальной форме), знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок.
2. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **0** и **1**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.
3. Входной язык содержит операторы условия **if ... then ... else** и **if ... then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<**, **>**, **=**, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).
4. Входной язык содержит операторы цикла **for (...; ...; ...) do ...**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).
5. Входной язык содержит операторы цикла **while (...) ... done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).
6. Входной язык содержит операторы цикла **do ... while (...)**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.
7. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, римских чисел, знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок. Римскими считать числа, записанные большими буквами **X**, **V** и **I**.
8. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **true** и **false**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.
9. Входной язык содержит операторы условия **if ... then ... else** и **if ... then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<**, **>**, **=**, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

10. Входной язык содержит операторы цикла **for (...; ...; ...) do ...**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

11. Входной язык содержит операторы цикла **while (...) ... done**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<**, **>**, **=**, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

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

13. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок. Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

14. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, символьных констант **'T'** и **'F'**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.

15. Входной язык содержит операторы условия **if ... then ... else** и **if ... then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<**, **>**, **=**, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

16. 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 (**:=**).

17. Входной язык содержит операторы цикла **do ... while (...)**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<=**, =**>**, **=**, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

18. Входной язык содержит операторы цикла **while (...) ... done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

19. Входной язык содержит арифметические выражения, разделённые символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, символьных констант (один символ в одинарных кавычках), знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок.

20. Входной язык содержит логические выражения, разделённые символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок. Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

21. Входной язык содержит операторы цикла **for (...; ...; ...) do ...**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

22. Входной язык содержит операторы цикла **do ... while (...)**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<=**, **=>**, **=**, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

23. Входной язык содержит операторы условия **if ... then ... else** и **if ... then**, разделённые символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения **<**, **>**, **=**, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

24. Входной язык содержит операторы цикла **while (...) ... done**, разделённые символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения **<**, **>**, **=**, шестнадцатеричные числа, знак присваивания (**:=**). Шестнадцатеричными числами считать последовательность цифр и символов **a**, **b**, **c**, **d**, **e**, **f**, начинающуюся с цифры (например, 89, 45ac, 0abc).

Контрольные вопросы

1. Какую роль выполняет лексический анализ в процессе компиляции?
2. Как связаны лексический и синтаксический анализ?
3. Какие проблемы необходимо решить при построении лексического анализатора на основе конечного автомата?
4. Чем отличаются таблица лексем и таблица идентификаторов? В какую из этих таблиц лексический анализатор не должен помещать ключевые слова, разделители и знаки операций?

Список литературы

1. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.
2. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p.
3. Fast Lexical Analyzer. Режим доступа: http://flex.sourceforge.net/.
4. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.