Лабораторная работа № 4
Проектирование лексического анализатора

Цель работы: изучение основных понятий теории регулярных грамматик, ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения сканера с помощью программного инструментария Flex на примере заданного простейшего входного языка.

Для выполнения лабораторной работы требуется написать программу на языке Flex, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде текстового файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

1. Краткие теоретические сведения

1.1. Лексический анализатор

Первая фаза компиляции называется лексическим анализом или сканированием. *Лексический анализатор* читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*.

Лексема (логическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем 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 передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора. Для каждой лексемы анализатор строит выходной *токен* (token) вида

⟨имя_токена, значение_атрибута⟩

Он передается последующей фазе, синтаксическому анализу. Первый компонент токена, имя_токена, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице символов, соответствующую данному токену. Информация из записи в таблице символов необходима для семантического анализа и генерации кода.

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 $\langle$**id**, 1$\rangle$, где **id** – абстрактный символ, обозначающий идентификатор, а 1 указывает запись в таблице символов для `a`. Запись таблицы символов для некоторого идентификатора хранит о нем такую информацию как его имя и тип.
2. Символ присваивания = представляет собой лексему, которая отображается в токен $\langle$=$\rangle$. 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 $\langle$**id**, 2$\rangle$, где 2 указывает на запись в таблице символов для `b`.

4. + является лексемой, отображаемой в токен $\langle + \rangle$.
5. c – лексема, отображаемая в токен $\langle \textbf{id}, 3 \rangle$, где 3 указывает на запись в таблице символов для c.
6. * – лексема, отображаемая в токен $\langle * \rangle$.
7. d – лексема, отображаемая в токен $\langle \textbf{id}, 4 \rangle$, где 4 указывает на запись в таблице символов для d.

Пробелы, разделяющие лексемы, лексическим анализатором пропускаются.

Представление инструкции присваивания после лексического анализа в виде последовательности токенов

$$\langle \textbf{id}, 1 \rangle \langle = \rangle \langle \textbf{id}, 2 \rangle \langle + \rangle \langle \textbf{id}, 3 \rangle \langle * \rangle \langle \textbf{id}, 4 \rangle .$$

В таблице 1 приведены некоторые типичные токены, неформальное описание их шаблонов и некоторые примеры лексем. *Шаблон* (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, в инструкции на языке программирования C

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

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
comparison	< или > или <= или >= или == или !=	<=, !=
id	Буква, за которой следуют буквы и цифры	pi, score, D2
number	Любая числовая константа	3.14159, 0, 6.02e23
literal	Все, кроме ", заключенное в двойные кавычки	"core dumped"

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 объем обрабатываемой информации, так как лексический анализатор структурирует поступающий на вход исходный текст программы и выкидывает всю незначащую информацию;
- для выделения в тексте и разбора лексем возможно 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 исходный 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, приведенной на рис.1.

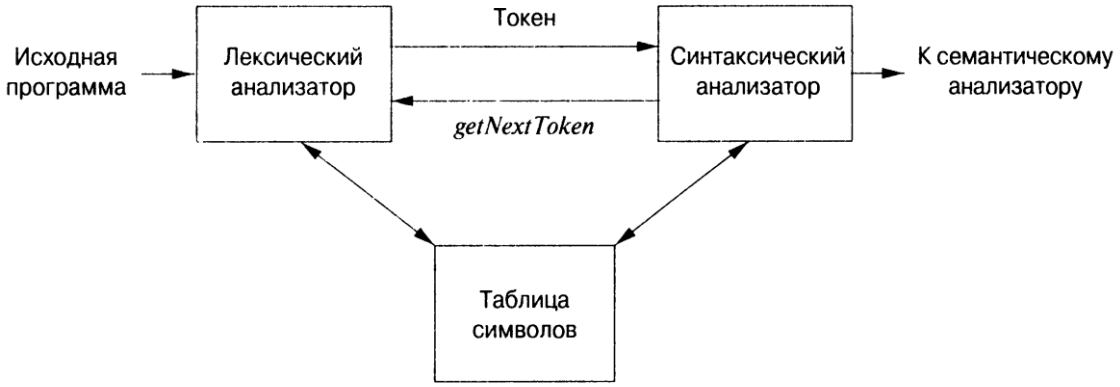

Рис.1. Взаимодействие лексического анализатора с синтаксическим.

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, поочередно обращаясь друг к другу. Лексический анализатор, найдя очередную лексему, передает 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. 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 ее можно представить как *таблицу лексем*, которая в каждой строчке должна содержать информацию о виде лексемы, ее типе и, возможно, значении. Обычно такая таблица имеет два столбца: первый – строка лексемы, второй – указатель на информацию о лексеме, может быть включен 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 этой таблице поле «значение» подразумевает некое кодовое значение, которое будет помещено в итоговую таблицу лексем. Значения, приведенные 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.

1.2. Генератор лексических анализаторов Flex

В лабораторной работе для реализации лексического анализатора будет использоваться программное средство Flex [1].

Программный инструментарий Lex (в более поздних реализациях, Flex) позволяет определить лексический анализатор, указывая регулярные выражения для описания шаблонов токенов [2]. Входные обозначения для Flex обычно называют *языком* Flex, а сам инструмент — *компилятором* Flex. Компилятор Flex преобразует входные шаблоны в диаграмму переходов и генерирует код (в файле с именем `lex.yy.c`), имитирующий данную диаграмму переходов.

Рис.2. Схема использования Flex

На рис. 2 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 компилятором Си в файл с названием `a.out`, как обычно. Выход компилятора Си представляет собой работающий лексический анализатор, который может получать поток входных символов и выдавать поток токенов.

Обычно скомпилированная программа на языке Си, которая на рис. 2 показана как `a.out`, 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 все 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 определение, 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 созданы многие 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.

Пример программы на языке Flex для подсчета 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++; }
%%
main(int argc, char **argv)
{
     yylex();
     printf("%8d%8d%8d\n", lines, words, chars);
}
```

В этом примере определены все три раздела программы на Flex.

В первом разделе объявлены три переменных-счетчика 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 счетчики для слов и символов. Массив символов `yytext` всегда содержит текст, соответствующий данному шаблону. В нашем случае он используется для расчета длины слова;
2) \n соответствует символу перевода строки. В случае совпадения входного потока с данным шаблоном происходит увеличение счетчиков 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 подсчета 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 3 перечислены специальные символы, использующиеся в регулярных выражениях (шаблонах) Flex.

Таблица 3. 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} соответствует повторению буквы A от одного до трех раз, а 0{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 свое значение и интерпретируются как два символа: \ и *.

2. Пример выполнения работы

2.1. Задание для примера

В качестве задания возьмем входной язык, содержащий набор условных операторов **if … then … else** и **if … then**, разделенные символом **;** (точка с запятой). Операторы условия содержат логические выражения, построенные с помощью круглых скобок и операций **or**, **xor**, **and**, операндами которых являются идентификаторы и целые десятичные константы без знака. В исполнительной части эти операторы содержат или оператор присваивания (**:=**), или другой условный оператор.

2.2. Грамматика входного языка

Описанный выше входной язык может быть построен с помощью КС-грамматики G({**if**, **then**, **else**, **a**, **:=**, **or**, **xor**, **and**, **(**, **)**, **;**}, {S, F, E, D, C}, **P**, S) с правилами **P**:

S → F; | SF;
F → **if** E **then** T **else** F | **if** E **then** F | **a :=** E
T → **if** E **then** T **else** T | **a :=** E
E → E **or** D | E **xor** D | D
D → D **and** C | C
C → **a** | **(**E**)**

Описание грамматики построено в форме Бэкуса-Наура.

2.3. Описание конечного автомата для распознавания лексем входного языка

Лексемами данного входного языка являются:

- шесть ключевых слов языка (**if**, **then**, **else**, **or**, **xor**, **and**);
- разделители (**(**, **)**, **;**);
- знак операции присваивания (**:=**);
- идентификаторы;
- целые десятичные константы без знака.

Для перечисленных типов лексем можно построить регулярную грамматику, а затем на её основе создать КА. Однако построенная таким образом грамматика, с одной стороны, будет элементарно простой, а с другой стороны – громоздкой и малоинформативной. Поэтому можно построить КА непосредственно по описанию лексем. Для этого нужно ещё описать идентификаторы и целые десятичные константы без знака:

- идентификатор – это произвольная последовательность малых и прописных букв латинского алфавита (от **A** до **Z** и от **a** до **z**), цифр (от **0** до **9**) и знака подчёркивания (**_**), начинающаяся с буквы или со знака подчёркивания;
- целое десятичное число без знака – это произвольная последовательность цифр (от **0** до **9**), начинающаяся с любой цифры.

Границами лексем будут служить пробел, знак 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.

На рис. 3 представлен фрагмент графа переходов КА, отвечающий за распознавание разделителей, знака присваивания, переменных и констант. Приняты следующие обозначения:

- П – любой незначащий символ (пробел, знак табуляции, перевод строки, возврат каретки);
- Б – любая буква английского алфавита (прописная или строчная) или символ подчёркивания;
- Б(*) – любая буква английского алфавита (прописная или строчная) или символ подчёркивания, кроме перечисленных в скобках;
- Ц – любая цифра от 0 до 9;
- F – функция обработки таблицы лексем, вызываемая при переходе КА из одного состояния в другое. Её аргументы:
    - v – переменная, запомненная при работе КА;
    - d – константа, запомненная при работе КА;
    - a – текущий входной символ КА.

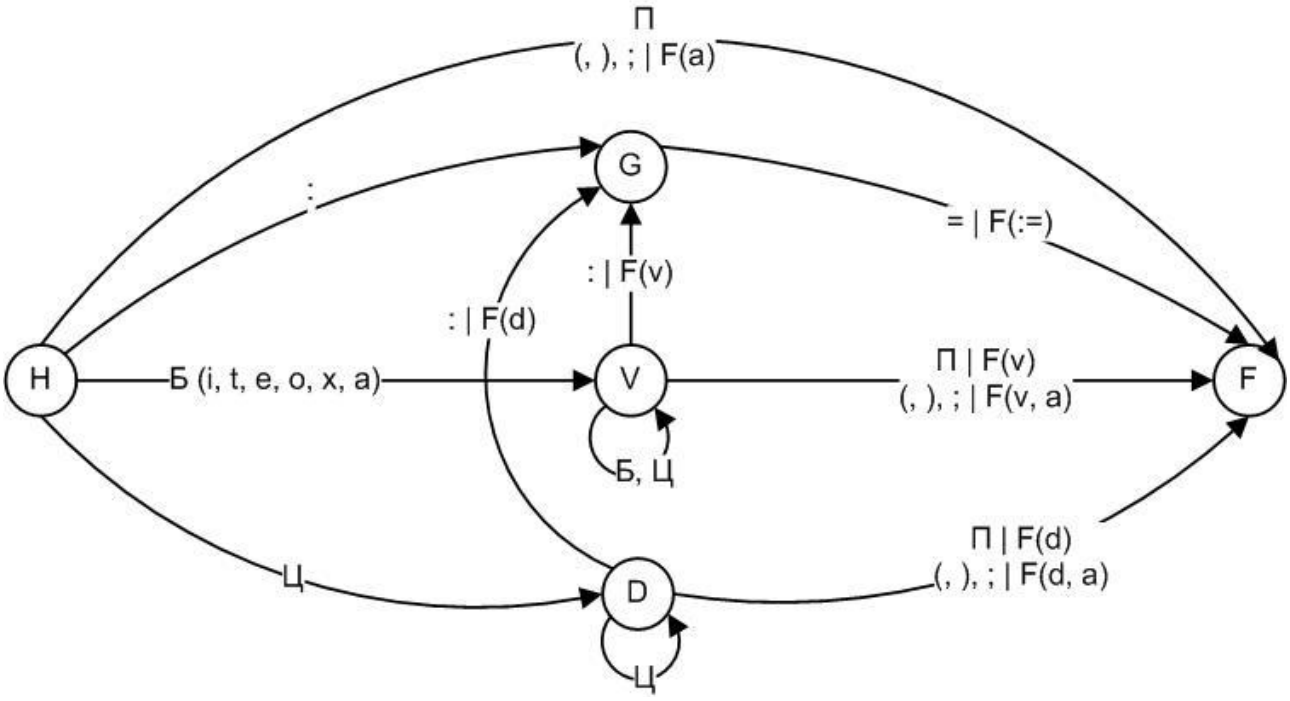

Рис. 3. Фрагмент графа переходов КА для распознавания лексем (кроме ключевых слов)

На рис. 4 изображён фрагмент графа переходов КА, отвечающий за распознавание ключевых слов **if** и **then** (в этом фрагменте указаны также ссылки на состояния,

изображённые на рис. 3). Аналогичные фрагменты могут быть построены и для других ключевых слов.

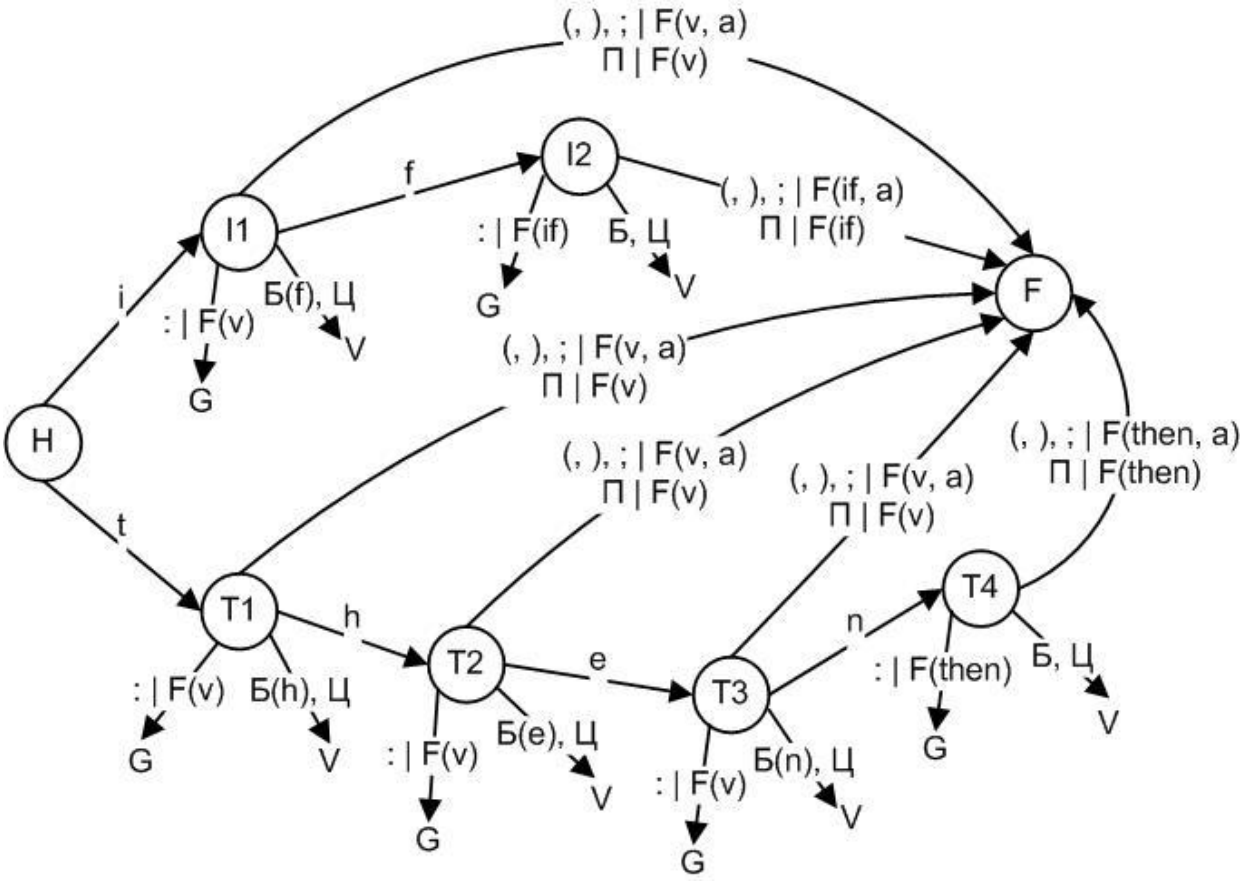

Рис. 4. Фрагмент графа переходов КА для ключевых слов **if** и **then**

С учётов введённых выше обозначений полностью КА можно описать следующим образом:

M (**Q**, **Σ**, δ, q_0, **F**)

Q = {H, G, V, D, I1, I2, T1, T2, T3, T4, E1, E2, E3, E4, O1, O2, X1, X2, X3, A1, A2, A3, F}

Σ = A (все допустимые алфавитно-цифровые символы)

q_0 = H

F = {F}

δ – функция переходов для этого КА.

Из начального состояния H литеры «i», «t», «e», «x», «o», «a» ведут в начало цепочек состояний, каждая из которых соответствует ключевому слова входного языка:

- состояния I1, I2 – ключевому слову **if**;
- состояния T1, T2, T3, T4 – ключевому слову **then**;
- состояния E1, E2, E3, E4 – ключевому слову **else**;
- состояния O1, O2 – ключевому слову **or**;
- состояния X1, X2, X3 – ключевому слову **xor**;
- состояния A1, A2, A3 – ключевому слову **and**.

Остальные литеры ведут к состоянию, соответствующему переменной (идентификатору), – V. Если в какой-либо из цепочек встречается литера, отличная от ключевого слова, или цифра, то КА тоже переходит в V.

Если встречается граница лексемы, то уже прочитанная часть ключевого слова запоминается как переменная (чтобы правильно выделять идентификаторы вида «i» или «els»).

Цифры ведут в состояние D. Состояние G соответствует знаку присваивания. В него КА переходит, получив на вход двоеточие, и ожидает в этом состоянии символа «=».

Состояние H – начальное состояние КА, а состояние F – его конечное состояние. КА работает с непрерывным потоком лексем, поэтому, перейдя в конечное состояние, он тут же должен возвращаться в начальное, чтобы распознавать очередную лексему.

На графе не обозначено состояние ER (ошибка), в которое КА должен переходить всегда, когда получает на вход символ, по которому нет переходов из текущего состояния.

Функция F, которой помечены дуги КА на графе, соответствует выполнению записи данных в таблицу лексем.

2.4. Реализация на языке Flex

```
%option noyywrap yylineno
%{
#include <stdio.h>
#include <string.h>

#define SIZE 6

char* filename;
char* keywords[SIZE] = {"if", "then", "else", "and", "or", "xor"};
%}

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

%%

({letter}|"_")({letter}|{digit}|"_")* {
     if(resWord(yytext))
     {
          printf("%s:%d KEYWORD %s\n", filename, yylineno,
                                                yytext);
     }else{
          printf("%s:%d IDENTIFIER %s\n", filename, yylineno,
                                                   yytext);
     }
}
{digit}+ {
     printf("%s:%d NUMBER %s\n", filename, yylineno, yytext);
}
":=" {
     printf("%s:%d ASSIGN %s\n", filename, yylineno, yytext);
}
{delim} {
     printf("%s:%d DELIMITER %s\n", filename, yylineno, yytext);
}
{ws}+ ;
```

```
. {
     printf("%s:%d Unknown character '%s'\n", filename, yylineno,
                                                        yytext);
  }

%%

int resWord(char* id)
{
     int i;
     for(i = 0; i < SIZE; i++)
     {
          if(strcmp(id, keywords[i]) == 0)
          {
               return 1;
          }
     }
     return 0;
}

int main(int argc, char** argv)
{
     if(argc < 2)
     {
          perror("Input file name is not specified");
          return 1;
     }
     yyin = fopen(argv[1], "r");
     if(yyin == NULL)
     {
          perror(argv[1]);
          return 1;
     }
     filename = strdup(argv[1]);
     yylineno = 1;
     yylex();
     return 0;
}
```

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 ее именем следует указать «no», как в случае с `noyywrap`. Полный список допустимых опций можно найти в документации по Flex [1, 3].

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. Использование `yywrap` имеет смысл тогда, когда в программе на Flex нет собственной функции `main`.

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 ее значение на 1, когда встречается символ '\n'. При этом Flex не инициализирует эту переменную. Поэтому в функции `main` перед вызовом функции лексического анализа `yylex` `yylineno` присваивается 1.

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

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

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

Входной текстовый файл содержит следующую программу, соответствующую правилам грамматики из раздела 2.2.

```
_A := 5;
B := 0;
if (_A and B) then
     _A := B
else if (_A xor B) then
     B := _A
else
     _A := _A or B;
```

Для компиляции и запуска программы используются следующие команды:

```
flex example.l
gcc lex.yy.c -lfl
./a.out
```

Результат выполнения:

```
program:1 IDENTIFIER _A
program:1 ASSIGN :=
program:1 NUMBER 5
program:1 DELIMITER ;
program:2 IDENTIFIER B
program:2 ASSIGN :=
program:2 NUMBER 0
program:2 DELIMITER ;
program:3 KEYWORD if
program:3 DELIMITER (
program:3 IDENTIFIER _A
program:3 KEYWORD and
program:3 IDENTIFIER B
program:3 DELIMITER )
program:3 KEYWORD then
program:4 IDENTIFIER _A
program:4 ASSIGN :=
```

```
program:4 IDENTIFIER B
program:5 KEYWORD else
program:5 KEYWORD if
program:5 DELIMITER (
program:5 IDENTIFIER _A
program:5 KEYWORD xor
program:5 IDENTIFIER B
program:5 DELIMITER )
program:5 KEYWORD then
program:6 IDENTIFIER B
program:6 ASSIGN :=
program:6 IDENTIFIER _A
program:7 KEYWORD else
program:8 IDENTIFIER _A
program:8 ASSIGN :=
program:8 IDENTIFIER _A
program:8 KEYWORD or
program:8 IDENTIFIER B
program:8 DELIMITER ;
```

3. Требования к оформлению отчета

Отчет должен содержать следующие разделы:

- Задание по лабораторной работе.
- Описание КС-грамматики входного языка в форме Бэкуса-Наура.
- Описание алгоритма работы сканера или граф переходов конечного автомата для распознавания цепочек (в соответствии с вариантом задания).
- Текст программы на языке Flex.
- Выводы по проделанной работе.

4. Варианты заданий

1. Входной язык содержит арифметические выражения, разделенные символом **;** (точка с запятой). Арифметические выражения состоят из идентификаторов, десятичных чисел с плавающей точкой (в обычной и логарифмической форме), знака присваивания (**:=**), знаков операций +, –, *, / и круглых скобок.

2. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **true** и **false**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.

3. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).

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

5. Входной язык содержит арифметические выражения, разделенные 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**.

6. Входной язык содержит логические выражения, разделенные символом **;** (точка с запятой). Логические выражения состоят из идентификаторов, констант **0** и **1**, знака присваивания (**:=**), операций **or**, **xor**, **and**, **not** и круглых скобок.

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

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

9. Входной язык содержит арифметические выражения, разделенные 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).

10. Входной язык содержит логические выражения, разделенные 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).

11. Входной язык содержит операторы условия **if … then … else** и **if … then**, разделенные 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).

12. Входной язык содержит операторы цикла **for (…; …; …) do**, разделенные 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).

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

14. Входной язык содержит логические выражения, разделенные 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**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

16. Входной язык содержит операторы цикла **for (…; …; …) do**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки

сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).

17. Входной язык содержит операторы цикла **while (…) … done**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).
18. Входной язык содержит операторы цикла **do … while (…)**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <=, =>, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (**:=**).
19. Входной язык содержит операторы цикла **while (…) … done**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).
20. Входной язык содержит операторы цикла **do … while (…)**, разделенные символом **;** (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <=, =>, =, строковые константы (последовательность символов в двойных кавычках), знак присваивания (**:=**).
21. Входной язык содержит операторы цикла **while (…) … done**, разделенные 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).
22. Входной язык содержит операторы цикла **do … while (…)**, разделенные 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 цикла **while (…) … done**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.
24. Входной язык содержит операторы цикла **do … while (…)**, разделенные символом **;** (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, римские числа, знак присваивания (**:=**). Римскими считать числа, записанные большими буквами **X**, **V** и **I**.

5. Основные контрольные вопросы

1. Что такое трансляция, компиляция, транслятор, компилятор?
2. Из каких процессов состоит компиляция? Расскажите об общей структуре компилятора.
3. Какую роль выполняет лексический анализ в процессе компиляции?
4. Как связаны лексический и синтаксический анализ?
5. Дайте определение цепочки, языка. Что такое синтаксис и семантика языка?
6. Какие существуют методы задания языков? Какие дополнительные вопросы необходимо решить при задании языка программирования?
7. Что такое грамматика? Дайте определения грамматики.
8. Как выглядит описание грамматики в форме Бэкуса-Наура.
9. Какие классы грамматик существуют? Что такое регулярные грамматики?

10. Дайте определения контекстно-свободной грамматики, выводимости цепочки, непосредственной выводимости, длины вывода.
11. Что такое конечный автомат? Дайте определение детерминированного и недетерминированного конечных автоматов.
12. Какие проблемы необходимо решить при построении сканера на основе конечного автомата?

6. Рекомендуемая литература

1. Fast Lexical Analyzer. Режим доступа: http://flex.sourceforge.net/
2. Ахо А.В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: Вильямс, 2008. – 1184 с. (раздел 3.5)
3. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.
4. Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: http://epaperpress.com/lexandyacc/download/lexyacc.pdf