**Лабораторная работа 3.**

**Построение лексического анализатора**

Первый этап компиляции программы – лексический анализ. Он заключается в том, что в тексте программы выделяются последовательные части (слова), которые называются ***лексемами***, несущие определенный смысл, но любая меньшая их часть всякого смысла совершенно лишена. Иными словами – это обозначения некоторых элементарных сущностей, который нельзя разделить на более мелкие части.

Основными видами лексем в подавляющем множестве языков программирования являются:

1. Служебные (ключевые) слова языка.

2. Арифметические, логические и прочие операции, отношения, присваивание.

3. Специальные символы (точка, точка с запятой, двоеточие, скобки и т.д.)

4. Идентификаторы.

5. Числовые константы.

6. Строковые константы.

Особняком надо отметить комментарии, которые лексический анализатор должен пропускать.

Отметим, что приведенное разделение лексем по видам не является, конечно, обязательным. Лексический анализатор вовсе не обязан разделять их на какие-то виды. Тем не менее обычно такая работа делается, во-первых, из-за того, что это делает сам лексический анализатор более понятным, во-вторых, это облегчает следующие этапы компиляции.

Посмотрим на примере простой программы на языке Паскаль, как может выглядеть результат ее лексического анализа.

**const** n=30;

**var** i,x:**integer**;

**begin** *{вычисление суммы членов арифметической прогрессии}*

x:=0;

**for** i:=1 **to** n **do** x:=x+i;

writeln(’Сумма = ’,x);

**end.**

Лексемами здесь являются:

**const**

n

=

30

;

**var**

i

,

x

:

**integer**

;

**begin**

x

:=

0

;

**for**

i

:=

1

**to**

n

**do**

x

:=

x

+

i

;

writeln

(

’Сумма = ’

***,***

x

)

;

**end**

**.**

Однако результатом лексического анализа должна быть более подробная информация и более тонкая работа. На этом этапе программа не просто разделяется на лексемы, а на самом деле заменяется потоком структур, которые несут информацию и о лексемах, и о некоторых иных сопутствующих и полезных вещах. Распространенной практикой при этом является построение таблиц лексем, в которых каждой лексеме присваивается номер (дескриптор).

Если речь идет о компиляции программы на Паскале, то предполагается, что некоторые таблицы лексем уже построены. Это, прежде всего служебные слова, затем разного рода операции и специальные символы.

Мы для иллюстрации предположим, что имеются следующие заранее построенные таблицы:

**Тип 1. Служебные слова**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | **const** |
| 2 | **var** |
| 3 | **integer** |
| 4 | **begin** |
| 5 | **for** |
| 6 | **to** |
| 7 | **do** |
| 8 | **end** |

**Тип 2. Операции и отношения**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | **=** |
| 2 | **:=** |
| 3 | **+** |

**Тип 3. Спец символы**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | **:** |
| 2 | ; |
| 3 | **.** |
| 4 | **,** |
| 5 | ( |
| 6 | ) |

Выбор именно таких элементов, понятно, просто иллюстрация к методу. В то же время анализатор, который мы построим, будет работать с любой программой, которая не использует служебных слов и прочих постоянных элементов Паскаля кроме тех, что попали в эти таблицы.

Таблицы идентификаторов и констант являются пустыми и заполняются по мере распознавания лексем. В нашем примере должно получиться

**Тип 4. Идентефикаторы**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | n |
| 2 | i |
| 3 | x |
| 4 | writeln |

и

**Тип 5. Числовые константы**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | 30 |
| 2 | 0 |
| 3 | 1 |

и

**Тип 6. Строковые константы**

|  |  |
| --- | --- |
| Номер | Лексема |
| 1 | ’Сумма = ’ |

Результатом работы лексического анализатора для нашего примера должна быть последовательность блоков, с информацией о лексемах. В самом простом варианте это:

<1,1>, <4,1>, <2,1>, <5,1>, <3,2>, <1,2>, <4,2>, <3,4>, <4,3>, <3,1>, <1,3>, <3,2>, <1,4>, <4,3>, <2,2>, <5,2>, <3,2>, <1,5>, <4,2>, <2,2>, <5,3>, <1,6>, <4,1>, <1,7>, <4,3>, <2,2>, <4,3>, <2,3>, <4,2>, <3,2>, <4,4>, <3,5>, <6,1>, <3,4>, <4,3>, <3,6>, <3,2>, <1,8>, <3,3>

где у пары <i,j> число i означает номер типа, число j – номер лексемы в типе.

Создание лексического анализатора начнем с конструирования автомата ***ALex*** со следующими свойствами:

**1. *ALex*** – детерминированный автомат.

**2.** Терминалами ***ALex*** являются все литеры с ASCII-кодами от 1 до 255.

**3. *Alex*** распознает цепочку тогда и только тогда, когда она является лексемой одного из перечисленных выше типов.

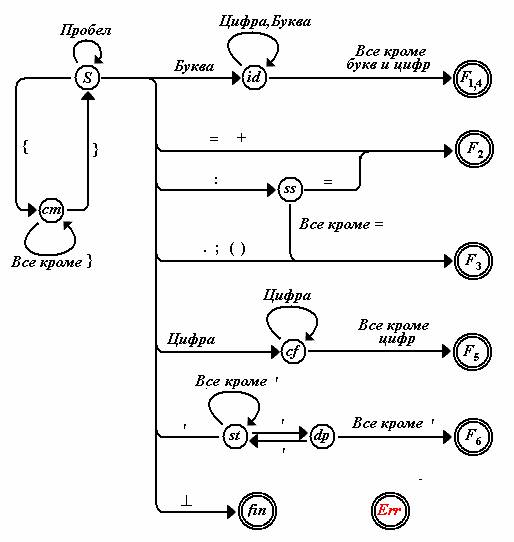
**4. Основными** конечными состояниями ***ALex*** являются {***F*1,4**, ***F*2**, ***F*3**, ***F*5**, ***F*6**}.

**5.** Когда на вход автомата подается еще не считанная часть программы, начинающаяся с очередной лексемы, то ***ALex*** переходит в одно из заключительных состояний ровно в тот момент, когда лексема полностью прочитана и, возможно, считан еще один символ. Для того, чтобы это выполнялось и для самой последней лексемы, добавим в текст программы, в самый ее конец, специальный заключительный символ ****.. Потребуем, чтобы переход в ***F*1,4** означал считывание лексемы типа 1 или типа 4. Иными словами автомат должен быть устроен так, что попадание в ***F*1,4** равносильно тому, что только что считано или ключевое слово, или идентификатор. Переход в ***F*2** должен означать считывание лексемы типа 2, переход в ***F*3** - лексемы типа 3,переход в ***F*5** -лексемы типа 5, переход в ***F* 6**-лексемы типа 6.

В условии 5 можно было бы типы 1 и 4 также разделить по разным конечным состояниям. Но соответствующий автомат будет слишком громоздок. А нам надо, чтобы использование автоматных конструкций помогало, а не мешало программированию. Так как длины некоторых идентификаторов или констант ничем не ограничены, конец считывания может произойти только при считывании одного лишнего символа. Эти неизбежные ситуации должны быть учтены в дальнейших построениях.

Вообще говоря, теория автоматов, согласно которой любой регулярный язык определяется конечным недетерминированным автоматом, оперирует с несколько иными определениями распознавания. Ценность этой теории видна при построении больших трансляторов, когда вручную ищутся регулярные грамматики (обычно в виде так называемых регулярных выражений) только для конкретных видов лексем, а вся остальная система строится с помощью подходящих конструкторов, проходя автоматическую процедуру минимизации.

Вот нужный нам автомат.



Состояние ***fin*** – состояние, в котором считывается последняя лексема. Состояние ***Err*** – состояние, в которое направлены все стрелки для всех терминалов и из всех состояний, которые на рисунке отсутствуют. Попадание в ***Err*** означает синтаксическую ошибку.

Процесс анализа будет идти путем циклического выполнения следующих шагов. Один шаг – это запуск автомата в состоянии S, когда на входе имеется еще не обработанная часть программы, начинающаяся очередной лексемой. Если автомат перейдет в ***Err***, то работа прекращается и выдается сообщение о синтаксической ошибке. Переход в состояние ***fin*** – конец лексического анализа. Переход в одно из оставшихся заключительных состояний – конец считывания очередной лексемы. После этого автомат “сбрасывается” в начальное состояние и, в зависимости от ситуации, последний считанный символ может быть возвращен во входной поток.

Понятно, что в процессе выполнения очередного шага надо выполнять определенную работу, благодаря которой возникают нужные нам таблицы и лексемы. Переходы автомата при этом играет роль указателей, какие действия и как надо делать. Чтобы написание программы анализатора по автомату стало почти автоматическим, нам осталось только каждому переходу приписать соответствующие действия. Эти действия следят за входящим и выходящим потоками и формируют новые записи в таблицах лексем.

Пусть **Buf** - буфер в который литера за литерой записывается считываемая лексема;

Действия:

***Pbuf*** – отправить очередную литеру в буфер **Buf**.

***RevLit*** – вернуть очередную литеру во входной поток.

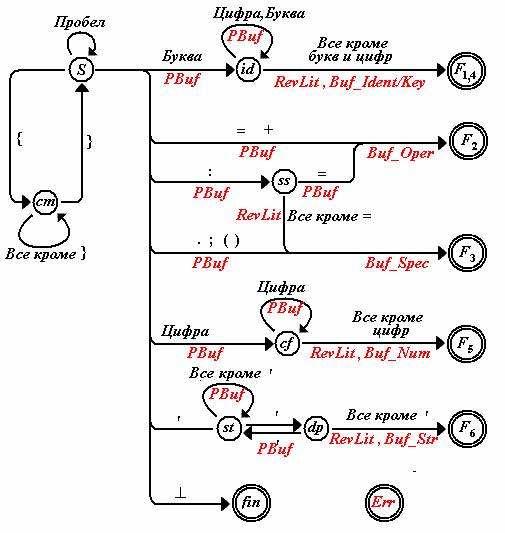
***Buf\_Intend/Key*** – определить, является ли содержимое буфера **Buf** ключевым словом. Если да, получить его тип и номер. Если нет, определить лежит ли **Buf** в таблице идентефикаторов. Если нет, то добавить. В любом случае вернуть тип и номер.

***Buf\_Oper*** – найти содержимое буфера **Buf** в таблице операций и вернуть тип и номер.

***Buf\_Spec*** – найти содержимое буфера **Buf** в таблице спецсимволов и вернуть тип и номер.

***Buf\_Num*** – добавить содержимое буфера **Buf** в таблицу числовых констант и вернуть тип и номер.

***Buf\_Str*** – добавить содержимое буфера **Buf** в таблицу строковых констант и вернуть тип и номер.



Пишем, наконец, основной модуль программы. Предполагаем, что в нашем распоряжении имеются функции, выполняющие выше перечисленные действия, программирование которых труда, конечно, не представляет.

**char** GetInputChar(); //выдает очередной символ компилируемой программы

//(очередной символ входного потока)

**void** RevLit(**char** c); //возвращает символ с во входной поток

**void** PutInBuf(**char** c); //добавляет символ с в буфер Buf

**bool** Number(**char** c); //возвращает true, если c – цифра 0..9

**void** Letter(**char** c); //возвращает true, если c – лат.буква или \_

**int** Buf\_Intend\_Key(**int** &Type); // определить, является ли содержимое

//буфера Buf ключевым словом.

//Если нет, определить лежит ли Buf

//в таблице идентификаторов. Если нет,

//то добавить. В любом случае в Type

// записать тип и вернуть номер.

**int** Buf\_Oper(**int** &Type); //найти содержимое буфера Buf в таблице

//операций и вернуть тип и номер.

**int** Buf\_Spec(**int** &Type); //найти содержимое буфера Buf в таблице

//спецсимволов и вернуть тип и номер.

**int** Buf\_Num(**int** &Type); //добавить содержимое буфера Buf в таблицу

//числовых констант и вернуть тип и номер.

**int** Buf\_Str(**int** &Type); //добавить содержимое буфера Buf в таблицу

//строковых констант и вернуть тип и номер.

**enume** State {S,cm,id,ss,cf,st,dp,fin} T;

**int** StepLexAnalize(**int** &Type) //Один шаг лексического

//анализатора

//Возвращает:

// 0 – конец потока.

//-1 – ошибка

// в остальных случаях –

// номер распозанной лексемы

// и в Type – ее тип

{

**char** c; //очередная литера

State T=S; //начальное значение очередного (текущего) состояния

**do// Инвариант цикла**

**// На вход автомата Alex подается**

**// компилируемая программа.**

**// Один оборот цикла соответствует**

**// одном шагу автомата Alex. В начале оборота:**

**// Значение переменной T – очередное состояние автомата.**

**// Остаток входного потока – еще не считанная цепочка.**

{

c=GetInputChar();

**switch**(T)

{

**case** S: **if**((c==’ ’)||(c==’\t’)||(c==’\r’)||(c==’\n’)) **break**;

**if**(c==’{’) {T=cm;**break**;}

**if**(c==’\’’){T=st;**break**;}

**if**(c==’****’) **return** 0;

//Выше перечислены случаи, когда с

//не отправляется в буфер

PutInBuf(c);

**if**(Letter(c)) {T=id; **break**;}

**if**((c==’=’)||(c==’+’)) **return** Buf\_Oper(Type);

**if**(c==’:’) {T=ss;**break**;}

**if**((c==’.’)||(c==’;’) ||(c==’(’)||(c==’)’))

**return** Buf\_Spec(Type);

**if**(Number(c)) {T=id; **break**;}

**return -1**; //Ошибка

**case** id:**if**(Number(c)||Letter(c)) {PutInBuf(c);**break;**}

**else** {RevLit(c);**return** Buf\_Intend\_Key(Type);}

**case** ss:**if**(c==’=’)

{

PutInBuf(c);

**return** Buf\_Oper(Type);

}

RevLit(c);

**return** Buf\_Spec(Type);

**case** cf:**if**(Number(c))

{

PutInBuf(c);

**break**;

}

RevLit(c);

**return** Buf\_Num(Type);

**case** st:**if**(c==’\’’) T=dp;

**else** PutInBuf(c);

**break**;

**case** dp:**if**(c==’\’’) { PutInBuf(c);T=st;}

RevLit(c);

**return** Buf\_Str(Type);

}

} **while true**;

}

//---------------------------

//Функция StepLexAnalize() используется в следующем

//основном модуле, которому требуется еще функция

//**void** PutInOut(**int** N,**int** T); отправляюшая пару

//<T,N> в результирующий поток лексем.

**int** LexAnalize(**int** &Type) //Лексический анализатор

{

**int** Type;

**int** N;

**do**

{

N=StepLexAnalize(Type);

**if**(N=<0) **return** N;

PutInOut(N,T);

} **while** **true**;

}

//-----------------------------

**Задание.**

Составить программу для лексического анализа текста программы

Выделять:

1. Служебные (ключевые) слова языка.

2. Арифметические, логические и прочие операции, отношения, присваивание.

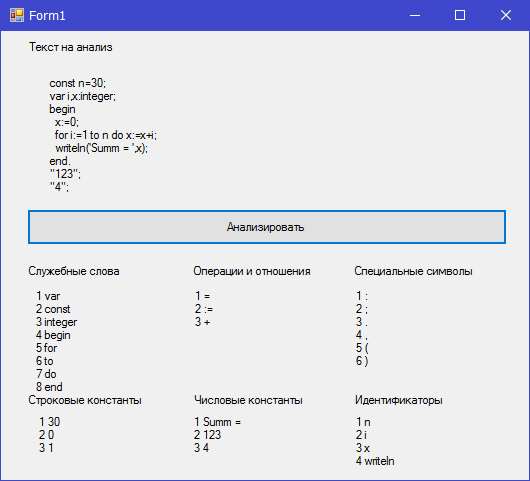
3. Специальные символы (точка, точка с запятой, двоеточие, скобки и т.д.)

4. Идентификаторы.

5. Числовые константы.

6. Строковые константы.

Примерный скриншот программы:



Результат должен выдаваться/обновляться при нажатии кнопки «Анализировать».

Анализатор настроить на язык С#.