آموزش Flex

با تشكر از آقاي مصطفي شاكري و خانم عاطفه عظيمي

Shakery.mostafa@gmail.com

بازنگري و آماده سازي مجدد : رضا فهيمي

rezafahimi@yahoo.com

✓ Flex چيست ؟

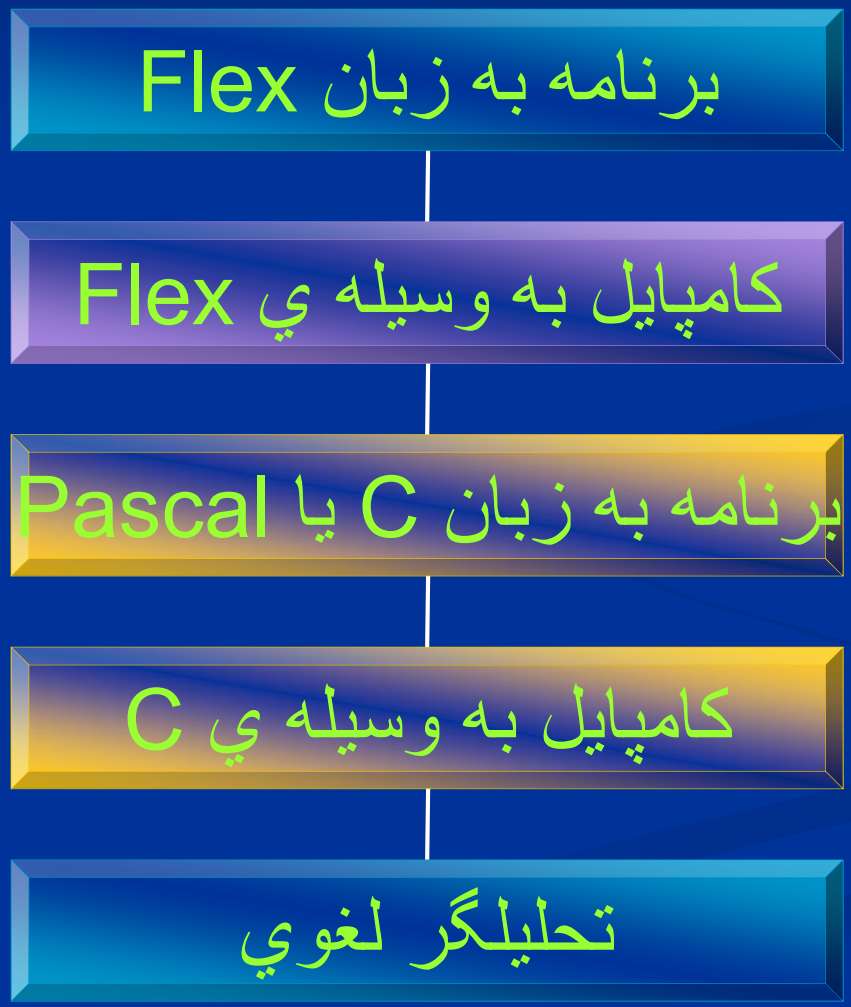
اصول و روش هاي ساخت تحليلگر لغوي براي زبان هاي مختلف يكسان است در نتيجه ابزارهايي ايجاد شده است كه با اخذ مشخصات لغوي زبان مبدا مي توانند تحليلگر لغوي آن را توليد كنند . اين گونه نرم افزار ها را توليد كننده ي تحليلگر لغوي مي نامند . يكي از مهمترين اين ابزار ها نرم افزار Flex (Fast lexical analyzer)است .

معمولا نرم افزار Flex به همراه نرم افزار ديگري به نام Bison استفاده مي شود .

ابزار Bison يك مولد تحليلگر نحوي (Syntax Analyzer) است كه در صورت استفاده در كنار Flex، ورودي خود را از آن گرفته و سپس عمليات تحليل معنايي را بر اساس قوانين تعريف شده ي گرامر انجام مي دهد.

علاوه بر نرم افزار هاي Flex و Bison , lex و yacc نيز وجود دارند كه در واقع Flex و Bison به ترتيب مشتق شده از Lex و Yacc هستند . Lex و Yacc در ابتدا با سيستم عامل UNIX كار مي كردند . Flex و Bison دوباره نوشته شده از Lex و Yacc براي DOS و به تازگي براي Windows32 مي باشند .

✓ مراحل استفاده از Flex جهت توليد تحليل گر لغوي :



✓نحوه ي اجراي Flex :

براي اجراي Flex ، دو راه وجود داره كه راه اول را به هيچ وجه توصيه نمي كنيم ولي جهت آشنايي بيان مي شود!

1) در روش اول كافيست بطور مستقيم فايل Flex.exe را از طريق خط فرمان (command prompt) اجرا كنيم و سپس گرامر را تا انتها بنويسيم ، و در پايان كليد Ctrl+Z (نشانگر پايان فايل) را بزنيم تا از ويرايشگر فرمان خارج شده و در صورت صحت گرامر فايل خروجي ايجاد شود .

2) در روش دوم ابتدا یک فایل (با پسوند L.) ایجاد کرده و سپس فایل را از طریق خط فرمان به کامپایلر Flex پاس مي دهيم و در نهايت در صورتي كه فايل مبداء وجود داشته و گرامر صحيح باشد فايل خروجي ايجاد خواهد شد .

بعد از انجام يكي از روش هاي بالا و در صورت نداشتن خطا در متن برنامه Flex فايلي با نام Yy.lex.c به عنوان خروجي و به زبان C در همان مسيري كه برنامه ي Flex وجود دارد ايجاد مي كند . كه براي ساخت فايل اجرايي بايد اين فايل را با كامپايلر C (Borlanc C) باز كرده و كامپايل كرد .

با اجراي فايل Yy.lex.c در Borlanc C ممكن است با خطا هايي مواجه شويم كه با رفع اين خطا ها و اجرا موفقيت آميز فايل اجرايي كه همان برنامه ي تحليلگر لغوي مي باشد ساخته مي شود .

حال براي استفاده از اين برنامه دو راه وجود دارد :

1) اجراي فايل اجرايي در خط فرمان و نوشتن متن در خط فرمان به عنوان ورودي و سپس ديدن جواب در همان خط فرمان .

2) نوشتن متن ورودي در يك فايل متني و اجراي فايل اجرايي در خط فرمان به همراه نام فايل ورودي و گرفتن يك فايل متني به عنوان خروجي .

در قسمت بعد انجام اين مراحل به صورت عملي بيان مي شود .

✓ کار با خط فرمان :

- اجراي Flex :

فايل flex.exe و فايل متني كه به زبان flex نوشته ايم و پسوند آن l. است را در مسيري مانند d:\ قرار مي دهيم و در command prompet (خط فرمان ويندوز) به روش زير عمل مي كنيم :

```
D:\flex    ÿ¿ÿ¿ÿ¿ÿ¿ÿ¿    ÿ¿ÿ¿ÿ¿ÿ¿ÿ¿
```

ÿ¿ÿ¿ÿ¿ÿ¿ÿ¿ : فايل متني به زبان Flex (با پسوند .L)

ÿ¿ÿ¿ÿ¿ÿ¿ÿ¿ : فايل متني به زبان C (با پسوند .c)

البته در شرايطي هم نمي توان نام فايل مقصد را وارد كرد و در اين حالت فايل مقصد با نام Yy.lex.c توليد خواهد شد !!!

به عنوان مثال :

```
C:\WINDOWS\system32\CMD.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\mostafa>d:

D:\>flex  input.l  output.c
```

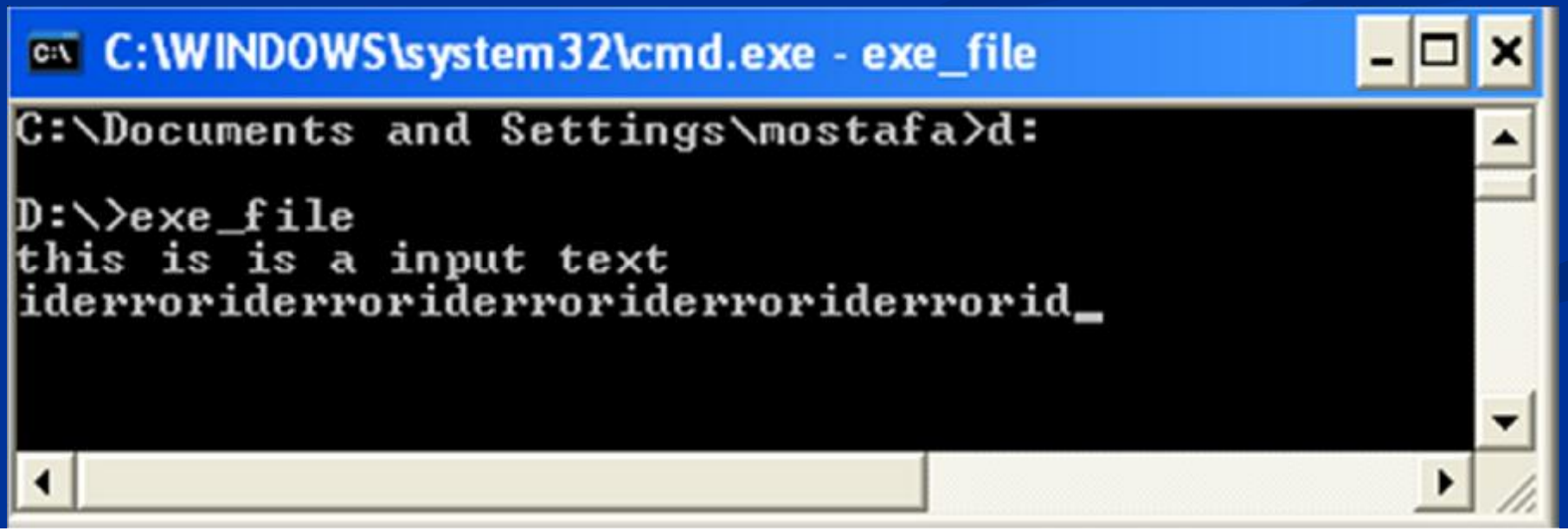
Flex برنامه ي مبدا (input.l) را خوانده و برنامه ي تحليلگر لغوي را به زبان c توليد كرده و در فايل مقصد (output.c) مي ريزد.

• اجراي تحليلگر لغوي :

روش اول (نوشتن متن ورودي در خط فرمان) :

```
D:\ЗYFпõH ÈPŭ
[ГÿHăЗкYHYΣüP εÿ
[ГÿHăΣüYHжч εÿ
```

به عنوان مثال :



Exe_file : فايل اجرايي (با پسوند .exe)

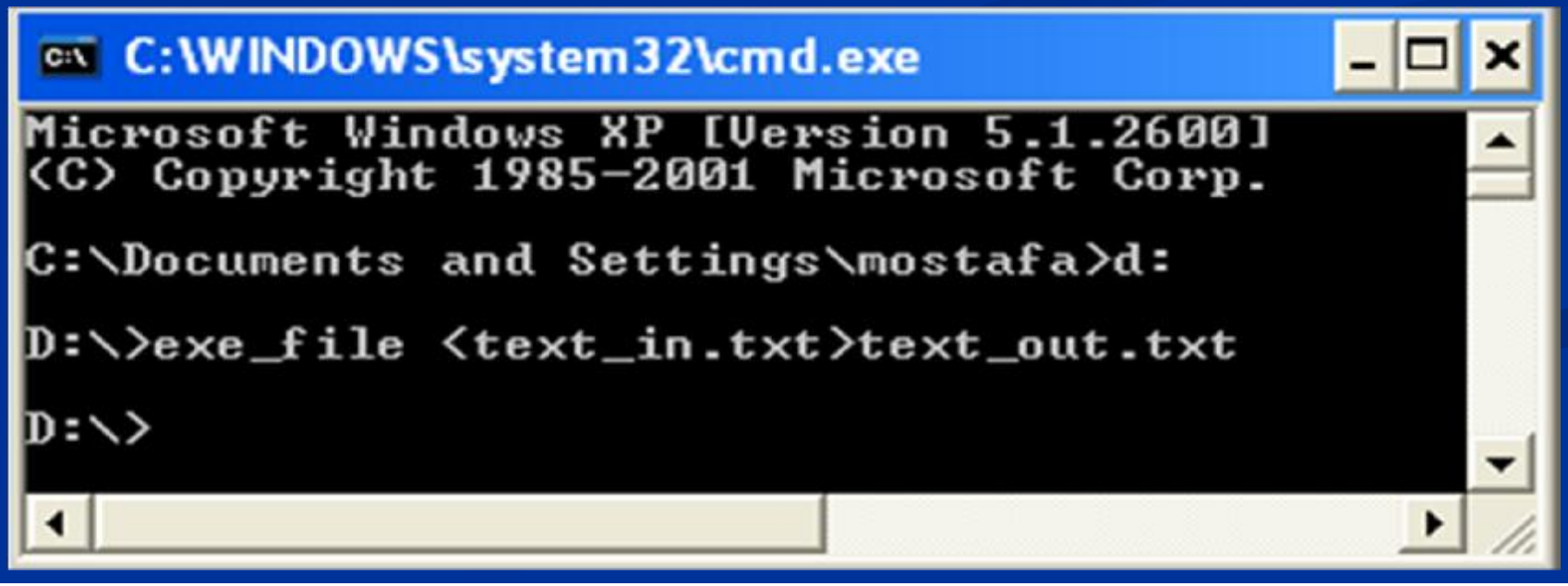
This is ... : متن به عنوان ورودي

Iderror ... : خروجي تحليلگر لغوي

روش دوم (نوشتن متن ورودي در فايل متني) :

D:\ ʒYƒ¤Ñ ¶ÈPŭ <ʒкYнY⊖ÉÑ¢Ÿ> ƐüYнж⊖ÉÑ¢Ÿ

به عنوان مثال :



Text_in.txt : فايل ورودي كه در آن همان متن روش اول نوشته شده است .

Text_out.txt : فايل خروجي كه در آن همان خروجي خط فرمان از روش اول نوشته شده است .

✓ آموزش زبان Flex :

متن درون فايل ورودي بايد به زبان flex باشد تا نرم افزار flex آن را بشناسد و خروجي اي به زبان c بسازد . Flex عبارات با قاعده را مي گيرد و تحليلگر لغوي مطابق آن را مي سازد . عبارت با قاعده را مي توان به صورت رشته به flex معرفي كرد.

براي مثال :

عبارت با قاعده ي if "if"

عبارت با قاعده ي while " while "

ساختار یک فایل flex :

تعاریف (definition)

%%

ترجمه ها (translation)

%%

توابع (function)

- **تعاريف :**

در قسمت تعاريف دو نوع اطلاعات قرار مي گيرد :

1) نام عبارت با قاعده

2)

```
%{
تعريف متغيرها
اعلان توابع
تعريف توابع
%}
```

چند مثال :

Lower [a-z]
Upper [A-Z]
Letter lower | upper

```
%{
#include<stdio.h>
Int nchar,nline;
%}
```

این قسمت به صورت کامل در فایلي که به زبان c توليد مي شود كپي مي شود

Digit [0-9]
Lower [a-z]
Upper [A-Z]
Letter lower | upper
Var { letter } | ({ letter } | { digit }) *

چند نکته :

این بخش اختياري است و مي توان برنامه اي نوشت که قسمت تعريف در آن نباشد. ولي استفاده از آن مي تواند باعث خوانا تر شدن برنامه شود براي مثال مي توان عبارت با قاعده اي را تعريف کرده و به آن نامي اختصاص داد و در قسمت هاي ديگر از آن نام استفاده کرد.

نحوه ي بيان عبارات باقاعده به زبان Flex در جدول صفحه ي بعد توضيح داده شده :

.	نشان دهنده ي هر كاراكتر به جز كاراكتر سر خط است
X	رشته هاي يك كاراكتري كه با كاراكتر منطبق هستندx
[مجموعه ي كاراكتر ها]	رشته هاي يك كاراكتري كه با هر يك از كاراكتر هاي مشخص منطبق شوند
[كاراكتر مقصد – كاراكتر مبدا]	يكي از كاراكتر هاي مبدا تا مقصد
r +	يكي الي چند تكرار از r
r ?	صفر الي يك تكرار از r
r *	صفر الي چند تكرار از r
r { m,n }	حد اقل m و حداكثر n تكرار از r

x|y	يعني يا x يا y
r *	صفر الي چند تكرار از r
xy	يعني اول x مي آيد و بعد y
x/y	رشته هايي كه قسمت اول آن با x و قسمت دوم آن با y منطبق باشد
[كاراكتر هاي مورد نظر^]	هر كاراكتر به جز كاراكتر هاي مشخص شده
\n	براي نشان دادن سر خط
\b	براي نشان دادن جاي خالي
\t	براي نشان دادن tab

(عملگر ها و كاراكتر ها)	اولويت با عبارات داخل پرانترز مي باشد و علاوه بر براي تركيب عملگر ها استفاده كرد مانند : (a|b)*
كاراكترهاي مورد نظر ^	كاراكتر هاي مورد نظري كه حتما در اول خر آمده باشند
{ }	براي اشتفاده ي كلماتي كه در قسمت تعاريف آمده اند مانند: Digit [0-9] Int ({ digit })+

- **ترجمه** (اين بخش اجباري است)

اين بخش مشخص مي كند هنگام كشف يك لغت مطابق يكي از عبارات با قاعده چه عملي بايد انجام شود.

ساختار ترجمه :

الگوي نشانه (عبارت با قاعده)	{ عمليات}

الگوي نشانه :عبارت با قاعده و يا يكي از اسامي تعريف شده در بخش تعاريف .

عمليات : دستورالعمل هايي به زبان c را نشان مي دهد كه هنگام يافتن دنباله اي از كاراكتر ها مطابق الگوي نشانه بايد اجرا شوند.

مثال :

تعاریف

```
Digit     [0-9]
Lower     [a-z]
Upper     [A-Z]
Letter    lower | upper
Var       { letter } | ( { letter } | { digit })*
Ws        [\n\t]+
%%
```

ترجمه

```
Ws        {}
"if"      { printf ( " I found "if" keyword " ) ; }
"else"    { printf ( " I found "else" keyword ) ; }
var       { printf ( "I found variable " ) ; }
%%
```

حتما باید آخرین خط نوشته شود (چرا؟)

ورودي: **If temp else if id34**

برنامه ي صفحه ي قبل

خروجي : **I found “if” keyword**

I found variable

I found “else” keyword

I found “if” keyword

I found variable

همان طور كه ملاحظه گرديد تحليلگر هاي لغوي ساخته شده با كشف هر نوع لغت پيغام مناسب چاپ مي كند . اما در يك كامپايلر واقعي تحليلگر لغوي به جاي چاپ يك پيغام بايد يك نشانه مشخص را براي تحليلگر نحوي ارسال كند . به اين منظور به جاي چاپ يك پيغام مي توان از دستور return استفاده كرد . در اين شرائط تحليلگر لغوي با كشف هر لغت نشانه مناسب را باز مي گرداند .

مثال :

```
%{
# define   ASSIGNMENT   300
# define   DIGIT   301
%}
%%
":="                    return(ASSIGNMENT );
[0-9]+                  return(DIGIT );
“a”                     return( 65 );
```

- **توابع** (اين قسمت اختياري مي باشد)

Flex تحليلگر لغوي (بخش ترجمه)را به يك تابع به نام **yylex()** تبديل مي كند .پس براي اجراي تحليلگر لغوي بايد اين تابع را فراخواني كرد.

مثال 1) برنامه اي به زبان **flex** بنويسيد كه تعداد كاراكتر ها و خطوط ورودي را شمارش كرده و نتيجه را چاپ كند.

(صفحه ي بعد)

```
%option noyywrap
%{
Int nchar , nline ;
%}
%%
[\n]   { nline++ ; }
.      { nchar++ ; }
%%
Int main ( void )
{
    yylex();
    printf( "%d  %d" , nchar , nline+1 );
   return (0) ;
}
```

به flex مي گويد كه فقط يك فايل ورودي وجود دارد

يعني هر كاراكتر به جز سرخط است

چرا با يك جمع كرديم؟

مثال 2) برنامه ي زير اغلب لغات برنامه به زبان پاسكال را شناسايي كرده و بر حسب مورد پيغام مناسب را چاپ مي كند.

```
Nquote [^’]
%%
[a-zA-Z] ( [a-zA-Z0-9] )*   printf ( “ID “) ;
“:=”                        printf ( “assignment ” ) ;
‘( { nquote } | “ )’        printf ( “charakter_string ” ) ;
“:”                         printf ( “colon ” ) ;
“,”                         printf ( “comma ” ) ;
[0-9]+                      printf ( “digseq ” ) ;
“.”                         printf ( “dot ” ) ;
“=”                         printf ( “egual ” ) ;
```

```
"("                    printf ( "lparen " ) ;
"<"                    printf ( "lt " ) ;
"<>"                   printf ( "notequal " ) ;
[0-9]+"."[0-9]+         printf ( "realnumber " ) ;
")"                    printf ( "rparen " ) ;
"^"                     printf ( "uparrow " ) ;
[\n\t\f]+              ;
%%
Int main()
{
    yylex() ;
Return 0 ;
}
```

نكات مهم اين مثال :

1) در اين برنامه كلمات كليدي توسط عبارت با قاعده ي [a-zA-Z] ([a-zA-Z0-9])* پذيرفته مي شوند.

2) در برخي موارد ممكن است ناسازگاري هايي بروز كند به عنوان مثال ممكن است بخش هاي مختلف يك دنباله توسط عبارات با قاعده مختلف پذيرفته شود به عنوان مثال فرض كنيد تحليلگر رشته ي "<>" را مي خواهد پردازش كند آيا وقتي كاراكتر اول را مي خواند ('<') پيام it چاپ مي كند يا هر دو كاراكتر را در كنار هم ("<>") پردازش كرده و پيام notequal چاپ مي شود ؟

“<” printf (“lt ”) ;

“<>” printf (“notequal ”) ;

دراين گونه موارد تحليلگر لغوي طولاني ترين رشته يعني <> را با توجه به عبارت با قاعده ي دوم مي پذيرد و notequal چاپ مي كند. به عبارتي مي توان گفت كل كاراكتر هاي بين دو علامت جداكننده (كه معمولا همان جاي خالي است) به عنوان يك رشته انتخاب مي شود و سپس پردازش مي شود.

علاوه بر yylex() توابعي ديگر هم به صورت پيش فرض در flex وجود دارد كه از آن جمله مي توان به موارد زير اشاره كرد :

Yytext() : رشته ي پذيرفته شده از ورودي در اين تابع قرار مي گيرد.

Yyleng() : طول رشته اي را كه در yytext قرار دارد ، نگه مي دارد.

input() : كاراكتر بعدي در متن را برمي گرداند .

output(c) : كاراكترc را در خروجي مي نويسد .

unput(c) : كاراكتر c را به متن پس مي فرستد تا بتواند توسط input() بعدي، خوانده شود .

yywrap() : هنگامي كه كار تحليل به پايان فايل رسيد ، توسط Flex فراخواني مي شود. اين تابع هميشه مقدار 1 را برمي گرداند

✓ کلمات کليدي :

براي معرفي کلمات کليدي به تحليلگر لغوي از طريق **flex** دو روش وجود دارد :

روش اول) هر کلمه ي کليدي را به عنوان يك نوع لغت درنظر گرفته و عبارت با قاعده ي مناسب را در ليست عبارات با قاعده درج مي کنيم .

مثال

```
...
%%
"program"                    printf ( "program" ) ;
"begin"                      printf ( "begin" ) ;
"end"                        printf ( "end" ) ;
[ a-zA-Z ] ( [ a-zA-Z0-9 ] )*   printf ( " id " ) ;
...
```

باید بعد از کلمات کليدي نوشته شود (چرا؟)

دقت : مكان تعريف عبارات با قاعده در برنامه مهم است . تحليلگر لغوي توليدي flex , مقايسه ي دنباله ي ورودي را با عبارات با قاعده به ترتيب از بالا به پايين انجام مي دهد . بنا بر اين عبارات با قاعده ي كلمات كليدي بايد قبل از عبارت با قاعده ي شناسه باشند در غير اين صورت تمام كلمات كليدي شناسه در نظر گرفته مي شوند.

روش دوم) كلمات كليدي را در يك جدول (جدول نماد ها) به عنوان مقدار اوليه وارد مي كنيم . هر گاه يك شناسه كشف مي شود اين شناسه در جدول جستجو مي شود اگر اين شناسه با كلمات كليدي جدول يكسان باشد شناسه يك كلمه ي كليدي است .

مثال :

```
%{
#include<string.h>
#include<iostream.h>
char c;
void toupper(char k[])
{
int i;
for(i=0;i<=strlen(k);++i)
 if(k[i]<='z' && k[i]>='a')
   k[i]- =32;
}
int is_keyword(char id[])
{
   char keyword [ 40 ][ 20 ] =
   { "AND","ARRAY","BEGIN","CASE","CONST",
    "IF","FOR","END","WHILE","THEN“ } ;
```

خروجي اين تابع رشته با حروف بزرگ مي باشد

```
  int i;
  For ( I = 0 ; I < 40 ; i++ )
     if ( strcmp ( id , keyword[ I ] ) == 0)
        return i;
  return -1;
}
%}
%%
[a-zA-Z]([a-zA-Z0-9])* { toupper( yytext );
                         If ( is_keyword( yytext ) != -1 )
                               cout << yytext ;
                         else
                               cout << "ID“ ;
                        }
[0-9]+"."[0-9]+          cout << "REALNUMBER “ ;
```

الگوريتم تشخيص
كلمات كليدي

```
"(*"                    do {
                            c=input () ;
                            if ( c == '*' )
                            {
                                c = input() ;
                                if ( c == ')' )
                                   break;
                                else
                                   unput(c) ;
                            }
                        } while ( 1 ) ;

"{"                     while ( ( c = input() ) != '}' ) ;
[\n\t]                  ;
.                       Printf ( “”) ‘\c’ : illegal character at
                        line %d ‘\n’ “ , yytext[0] , line_no);
%%
int main()
{
  yylex() ;
  return 0 ;
}
```

مثال : تحليلگري كه به حروف كوچك و بزرگ حساس نيست.

```
%{
#include <stdio.h>
int line_no=1;
%}
A [aA]
B [bB]
C [cC]
.
.
.
X [xX]
Y [yY]
Z [zZ]
%%
{A}{N}{D}                    return ( AND ) ;
{D}{O}                       return ( DO ) ;
[a-zA-Z]([a-zA-Z0-9])+       return ( IDENTIFIER ) ;
```

"(*" | "{"

```
{
  register int c;
  while ( (c = input() ) )
  {
    if ( c == '}' )
      break;
    else if ( c == '*' )
    {
      if ( ( c = inpute() ) == ')' )
        break;
      else
        unput(C) ;
    }
    else if ( c == '\n' )
      line_no ++ ;
    else if ( c == 0 )
      commenteof() ;
  }
}
;
```

[\t\f]

نوع register باعث مي شود كه يك كپي از اين متغير در cache قرار بگيرد تا سرعت عمليات بالا رود كه براي متغير هاي پر كاربرد مفيد است

كاراكتر را در نوع int ريخته ايم اما برنامه error نمي دهد (چرا؟)

```
\n                    line_no++;
%%
commentof()
{
  printf ( %d\n , line_no ) ;
  exit (1) ;
}
yywrap()
{
Return (1) ;
}
```

مثال : تعداد کاراکتر و خط

```
%option noyywrap
int num_lines=0;num_chars=0;
%%
\n   ++num_lines;++num_chars;
.    ++num_chars;
%%
Main ( int  argc , char  **argv ) {
 + + argv , - - argc;
If ( argc == 0 )
 yyin = fopen ( argv [ 0 ] , "r“ ) ;
else
 yyin = stdin ;
yylex() ;
printf("# of lines=%d,# of  chars = %d\n“ , num_lines ,num_chars ) ;
}
```

تعداد پارامتر هاي ورودي

اشاره گر به اول آرايه ي پارامتر هاي ورودي

در صورت نبود پارامتر ورودي اطلاعات از روي فايل خوانده مي شود

در صورت وجود پارامتر ورودي اطلاعات از صفحه كليد خوانده مي شود

مثال :

```
%{
int mylineno=0;
void handle comments(void);
%}
string      \"[^\n"]+\"
char        \'[^\n']+\'
prepro      ^#.*
ws          [\t]+
alpha       [A-Za-z]
dig         [0-9]
name        {alpha}({alpha}|{dig})*
int_num     [-+]?{dig}
num1        [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2        [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
Flt_num     { num1 } | { num2 }
```

يعني اگر اول خطي با # شروع شده بعد از آن هرچه كاراكتر آمده بود عملياتي انجام نده (مورد استفاده براي دستورات #include <...>)

يعني وقتي به // رسيدي بعد از آن هر كاراكتري و به هر تعداد بود (به جز كاراكتر خط جديد) براي آن هيچ عملياتي انجام نده

```
%%
{ws}
{prepro}
"//".*
"/*"          { handle_comments() ; }
\n            { mylineno++ ; }
{int_num}     { printf( "integer number[%s]\n“ , yytext ) ; }
{flt_num}     { printf("floatin-point number[%s]\n“ , yytext ) ; }
{name}        { printf ( "name[%s]\n“ , yytext ) ; }
{string}      { printf ( "string[%s]\n“ , yytext ) ; }
.             { printf ( "don't recognise[%s]\n“ , yytext ) ; }
```

```
%%
const char *keywords[ ] = {
"auto","break","char","const","sewitch","sizeof","
static","long“ };

const int keywords_longth = sizeof(keywords) / sizeo(keywords [ 0 ] );

int main()
{
  printf ( "**STARTING LEXICAL ANALYSIS**\N“ ) ;
  while ( yylex() != 0 ) ;
  printf ( "**FINISHED LEXICAL ANALYSIS**\N“ ) ;
  return 0 ;
}
```

```
void handle_comments(void)
{
int c;
While ( ( c = input() ) != 0 )
 {
  if ( c == '\n' )
    ++ mylineno ;
  else if ( c == '*' )
  {
    if ( ( c = input() ) == '/' )
      break ;
    else
      unput(c) ;
  }
 }
}
```

اين تابع در صورت ورود به قسمت توضيحات تا پايان آن پيش مي رود و پردازش خاصي انجام نمي دهد فقط اگر خط جديدي ايجاد شود به شمارنده ي خط يك واحد اضافه مي كند