

LEX/Flex Scanner Generator

flex - Fast Lexical Analyzer Generator

We can use **flex**^a to automatically generate the **lexical analyzer/scanner** for the lexical atoms of a language.

^aThe original version was known as **lex**.

Input

- The input to the **flex** program (known as **flex compiler**) is a set of **patterns** or **specification** of the **tokens** of the source language.
- **Actions** corresponding to different matches is also specified.

Input

- A pattern for every token class is specified as an extended regular expression.
- Corresponding action is a piece of C or C++ code.
- The specification can be written in a file (by default it is `stdin`).

Output

- The **flex** software compiles the **specification** (regular expression) to a DFA like object.
- It implements as a C or C++ program with the **action codes** corresponding to different patterns embedded in it.

Output

- The output of `flex` is a C file `lex.yy.c` with the main function `int yylex(void)`^a.
- This function, when called, returns a `token`.
- Attribute information of a `token` is available through the global variable `yyval`.

^aIt is possible to generate `c++` file `lex.yy.cc` with the option `%option c++`

flex Specification

The flex specification is a collection of **regular expressions** and the corresponding actions as C (C++) code. It has **three** parts:

Definitions

%%

Rules

%%

User code

flex Specification

- The **definition** has two parts. The portion within the pair of special parentheses **%{** and **%}** is copied verbatim in **lex.yy.c**.
- Similarly the **user code** is also copied.
- The other part of the **definition** contains regular name definitions, start conditions and other options.

flex Specification

- The **rules** are specified as **<pattern>** **<action>** pairs.
- The **patterns** are **extended regular expressions** corresponding to different **tokens**.
- Corresponding actions are given as C or C++ code. An action is taken at the end of the match of a pattern.

An Example: `lex.l`

```
/*  
 * A scanner for a toy language lex.l  
 */  
  
%{ // Copied verbatim  
  
#include <string.h>  
#include <stdlib.h>  
#include "y.tab.h" // tokens are defined  
int line_count = 1; yylType yylval;  
  
%}
```

```
%option noyywrap
```

```
%x CMNT INSTR
```

```
DELIM      ([ \t])
```

```
WHITESPACES ({DELIM}+)
```

```
P_DIG      ([1-9])
```

```
DIG        (0|{P_DIG})
```

```
NN_INT     (0|{P_DIG}{DIG}*)
```

```
LOWER      ([a-z])
```

```
LETTER     ({LOWER}|[:upper:])
```

```
IDENTIFIER (({LETTER}({LETTER}|{DIG})*)|(_+({LETTER}|{DIG})))
```

```
%%
```

```
"/*"          {BEGIN CMNT;}
<CMNT> .      {;}
<CMNT> \n      {++line_count;}
<CMNT> "*/"    {BEGIN INITIAL;}
\n            {
                ++line_count;
                return (int)'\n';
            }
"\".\".*\".\"\" {
    yylval.string=(char *)malloc((yyleng+1)*(sizeof(char)));
    strncpy(yylval.string, ytext+1, yytext-2);
    yylval.string[yytext-2]='\0';
    return STRNG;
}
```

```
{WHITESPACES} { ; }  
if           {return IF;}  
else        {return ELSE; }  
while       { return WHILE; }  
for         { return FOR; }  
int         { return INT; }  
\(  
\)         { return (int)'('; }  
\)         { return (int)')'; }  
\{         { return (int)'{'; }  
\}         { return (int)'}'; }  
;          { return (int)';'; }  
,         { return (int)','; }  
=          { return (int)=''; }  
"<"       { return (int)'<'; }
```

```
"<="      { return LEQ; }
"&"       { return (int)'&'; }
"+"       {
            yylval.integer = (int)'+';
            return BIN_OP;
        }
"_"       {
            yylval.integer = (int)'-';
            return BIN_OP;
        }
"*"       {
            yylval.integer = (int) '*';
            return BIN_OP;
        }
```

```
"/"      {
        yylval.integer = (int)'/';
        return BIN_OP;
    }

{IDENTIFIER} {
    yylval.string=(char *)malloc((yyleng+1)*(sizeof(char)));
    strncpy(yylval.string, yytext, yylen);
    yylval.string[yyleng]='\0';
    return ID;
}

{NN_INT}  {
        yylval.integer = atoi(yytext);
        return NUM;
    }
```

```
.          { printf("Invalid symbol %s at line %d\n",  
                                     yytext, line_count); }  
  
%%  
/* The function yywrap() */  
//int yywrap(){return 1;}
```

y.tab.h

```
#ifndef _Y_TAB_H
#define _Y_TAB_H

#define IF      300
#define ELSE    301
#define WHILE   302
#define FOR     303
#define INT     304
#define ID      306
#define NUM     307
#define STRNG   308
#define BIN_OP  310
```

```
#define LEQ 312

int yylex(void);
typedef union {
    char *string;
    int integer;
} yylType;

#endif
```

myLex.c

```
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
extern yylType yylval;
extern int yylex();
int main() // myLex.c
{
    int s;

    while((s=yylex()))
    switch(s) {
        case '\n': printf("\n");
```

```
        break;
case '(' : printf("<( > ");
        break;
case ')' : printf("<)> ");
        break;
case '{' : printf("<{> ");
        break;
case '}' : printf("<}> ");
        break;
case ';' : printf("<;> ");
        break;
case ',' : printf("<,> ");
        break;
case '=' : printf("<=> ");
```

```
                break;
case '<' : printf("<<> ");
                break;
case '&' : printf("<&> ");
                break;
case LEQ : printf("<<=> ");
                break;
case BIN_OP : printf("<BIN_OP, %c> ",
                    (char) yylval.integer);
                break;
case IF    : printf("<if> ");
                break;
case ELSE  : printf("<else> ");
                break;
```

```
case WHILE : printf("<while> ");
             break;
case FOR   : printf("<for> ");
             break;
case INT   : printf("<int> ");
             break;
case ID    : printf("<ID, %s> ", yylval.string);
             free (yylval.string);
             break;
case NUM   : printf("<NUM, %d> ",yylval.integer);
             break;
case STRNG : printf("<STRNG, %s> ",
                    yylval.string);
             free (yylval.string) ;
```

```
        break;  
    default:  ;  
}  
return 0;  
}
```

input

```
/* Scanner input */  
int main()  
{  
    int n, fact, i;  
  
    scanf("%d", &n);  
    fact=1; @  
    for(i=1; i<=n; ++i) fact = fact*i;  
    printf("%d! = %d\n", n, fact);  
    return 0;  
} /* End */
```

How to Compile?

```
$ flex lex.l  $\Rightarrow$  lex.yy.c
$ cc -Wall -c lex.yy.c  $\Rightarrow$  lex.yy.o
$ cc -Wall -c myLex.c  $\Rightarrow$  myLex.o
$ cc myLex.o lex.yy.o  $\Rightarrow$  a.out
$ ./a.out < input > out
$
```

A Simple Makefile

```
objfiles = myLex.o lex.yy.o
a.out : $(objfiles)
    cc $(objfiles)
myLex.o : myLex.c
    cc -c -Wall myLex.c
lex.yy.c : lex.l y.tab.h
    flex lex.l
lex.yy.o :
clean :
    rm a.out lex.yy.c $(objfiles)
```

Output

```
<int> <ID, main> <( > <)>
```

```
<{>
```

```
<int> <ID, n> <,> <ID, fact> <,> <ID, i> <;>
```

```
<ID, scanf> <( > <STRNG, %d> <,> <&> <ID, n> <)> <;>
```

```
<ID, fact> <=> <NUM, 1> <;> Invalid symbol @ at line 6
```

```
<for> <( > <ID, i> <=> <NUM, 1> <;> <ID, i> <<=><ID, n> <;>
```

```
<ID, printf> <( > <STRNG, %d! = %d\n> <,> <ID, n> <,> <ID, f
```

```
<ID, return> <NUM, 0> <;>
```

```
<}>
```

Note

There is a problem with the rule for **comment**.

It will give wrong result for the following

`comment (input4)`

```
/* There is a string "within */ this  
    comment" and it will not work */
```

Try with the rule in **lex4.1**

Note

When the function `yyllex()` is called, it reads data from the input file (by default it is `stdin`) and returns a token. At the end-of-file (`EOF`) the function returns `zero (0)`.

Input/Output File Names

The name of the input file to the scanner can be specified through the global file pointer (FILE *) `yyin` i.e.

```
yyin = fopen("flex.spec", "r");
```

Similarly, the output file of the scanner^a can also be supplied through the file pointer `yyout` i.e.

```
yyout = fopen("lex.out", "w");
```

^aNote that it will affect the output of `ECHO` and not change the output of `printf()` etc.

Pattern and Action

A **pattern** must start from the **first column** of the line and the **action** must start from the **same line**. Action corresponding to a pattern may be empty.

The special parenthesis **%{** and **%}** of the definition part should also start from the **first column**.

Matching Rules

If more than one pattern matches, then the longest match is taken. If both matches are of same length, then the earlier rule will take precedence. In our example `if`, `else`, `while`, `for`, `int` are specified above the `identifier`. If we change the order, the keywords will not be identified separately.

Matching Rules

In our example ‘++’, is treated as two symbols ‘+’ and ‘+’. But if there is a rule for ‘++’, a single token can be generated.

Lexeme

If there is a match with the input text, the matched **lexeme** is pointed by the **char** pointer **yytext** (global) and its length is available in the variable **yyleng**.

The **yytext** can be defined to be an array by using **%array** in the **definition** section. The array size will be determined by **YYLMAX**.

The word "ECHO" is displayed in a red, serif font, enclosed within a white rectangular box with a thin blue border. This box is positioned on top of a solid dark blue rectangular background.

The command **ECHO** copies the text of **yytext** to the output stream of the scanner^a. The **default rule** is to **echo** any unmatched character to the output stream. (**lex1.1**)

^aBy default it is **stdout** and it can be changed by the file pointer **yyout**.

```
yymore(), yyless(int)
```

- `yymore()`: the **current lexeme** remains in the `yytext`. The **next lexme** is appended to it.
- `yyless(n)`: the scanner puts back all but the first *n* characters in the input buffer. Next time they will be rescanned. Both `yytext` and `yy leng` (*n* now) are properly modified. (`lex1.1`)

```
input(), unput(c)
```

`input()`: reads the next character from the input stream.

`unput(c)`: puts the character `c` back in the input stream^a.

^aflex manual: An important potential problem when using `unput()` is that if you are using `%pointer` (the default), a call to `unput()` destroys the contents of `ytext`, starting with its rightmost character and devouring one character to the left with each call. If you need the value of `ytext` preserved after a call to `unput()` (as in the above example), you must either first copy it elsewhere, or build your scanner using `%array` instead.

Start Condition

There is a mechanism to put a **conditional guard** for a rule. Let the condition for a rule be $\langle C \rangle$. It is written as

$\langle C \rangle P \{ \text{action} \}$

The scanner will try to match the input with the pattern P , only if its **start condition** is C .

In the example we have three rules guarded by the **start condition** $\langle \text{CMNT} \rangle$ used to remove the comment of a C program without any action.

Start Condition

A start condition is activated by a **BEGIN** action. In our example the start condition begins (**BEGIN CMNT**) after the scanner sees the starting of a comment (**/***).

Start Condition

The state of the scanner when no other start condition is active is called **INITIAL**. All rules without any start condition or with the start condition **<INITIAL>** are active at this state.

The command **BEGIN(0)** or **BEGIN(INITIAL)** brings the scanner from any other state to this state.

In our example the scanner comes back to **INITIAL** after consuming the comment.

Start Condition

Start conditions can be defined in the **definition** part of the specification using **%s** or **%x**. The start condition specified by **%x** is called an **exclusive** start condition. When such a start is active, only the rules guarded by it are activated. On the other hand **%s** specifies a an **inclusive** start condition.

In our example, only three rules are active after the scanner sees the beginning of a comment (**/***).

Start Condition

The scope of start conditions may be **nested** and the start conditions can be stacked. Three routines are of interest:

```
void yy_push_state(int new_state),  
void yy_pop_state() and  
int yy_top_state().
```

`yywrap()`

Once the scanner receives the **EOF** indication from the **YY_INPUT** macro (zero), the scanner calls the function **yywrap()**. If there is no other input file, the function returns **true** and the function **yylex()** returns zero (0).

But if there is another input file, **yywrap()** opens it, sets the file pointer **yyin** to it and returns **false**. The scanner starts consuming input from the new file.

```
%option noyywrap
```

This option in the definition makes scanner behave as if `yywrap()` has returned `true`. No `yywrap()` function is required.

Prototype of Scanner Function

The name and the parameters of the scanner function can be changed by defining `:YY_DECL` macro. As an example we may have

```
#define YY_DECL int scanner(vP) yylval *vP;
```

Architecture of Flex Scanner

The **flex compiler** includes a **fixed program** in its scanner. This program simulates the DFA. It reads the input and simulates the state transition using the state transition table constructed from the **flex specification**. Other parts of the scanner are as follows:

Architecture of Flex Scanner

- The transition table, start state and the final states - this comes from the construction of the DFA.
- Declarations and functions given in the definition and user code of the specification file - these are copied verbatim.

Architecture of Flex Scanner

- Actions specified in the rules corresponding to different patterns are put in such a way that the simulator can initiate them when a pattern is matched (in the corresponding final state).

Bibliography

References

[Flex1] <https://westes.github.io/flex/manual/>

[Flex2] <https://epaperpress.com/lexandyacc/download/flex.pdf>

[Flex3] https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_toc.html