

Lexical Analysers generation Tools

Flex

Introduction

FLEX (fast lexical analyzer generator), is a tool/computer program for generating lexical analyzers (scanners or lexers).

History

Lex

Lex is a popular scanner (lexical analyzer) generator Developed by M.E. Lesk and E. Schmidt of AT&T Bell Labs, and described in 1975. It is commonly used with the yacc parser generator.

It is the standard lexical analyzer generator on many Unix systems, and an equivalent tool is specified as part of the POSIX-standard.

Though originally distributed as proprietary software, some versions of Lex are now open source. Open source versions of Lex, based on the original AT&T code are now distributed as a part of open source operating systems such as OpenSolaris. One popular open source version of Lex, called **flex**, or the "fast lexical analyzer", is not derived from proprietary coding.

Flex

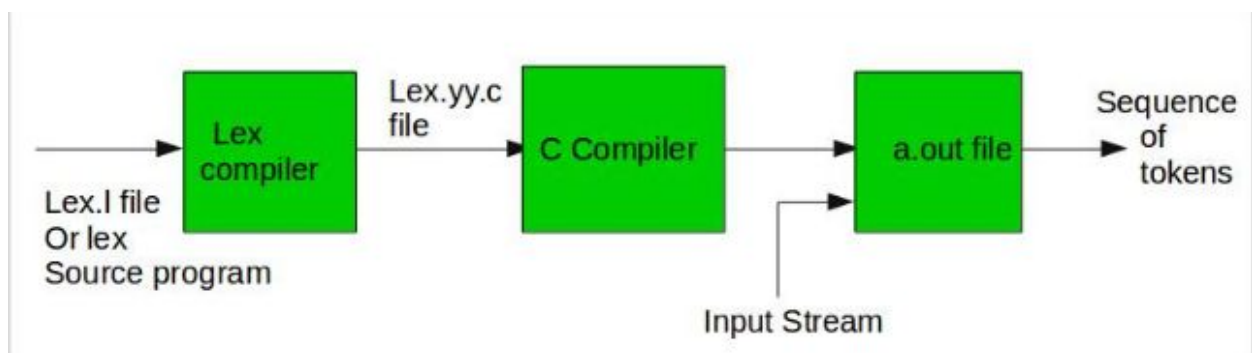
Flex was written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator.

Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

Functionality

Bison produces parser from the input file provided by the user.

The function **yylex()** is automatically generated by the flex when it is provided with a **.l file** and this **yylex()** function is expected by parser to call to retrieve tokens from current token stream. It's also is the main flex function which runs the Rule Section and extension (.l) is the extension used to save the programs.



Phases

phase 1: An input file describes the lexical analyzer to be generated named lex.l\lex is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.

phase 2: The C compiler compile lex.yy.c file into an executable file called a.out.

phase 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program structure

1. Definitions

```
%{  
    // Definitions  
%}
```

The definitions section contains the declaration of variables, regular definitions, manifest constants.

Example :

```
%{  
    int count = 0;  
%}
```

2. Rules

```
%%  
pattern action  
%%
```

The rules section contains a series of rules in the form: *pattern action* and pattern must be unintended and action begin on the same line in {} brackets.

Example :

```
%%  
    [A-Z] {Action code;}  
%%
```

3. User code

- a. This section contain C statements and additional functions.
- b. We can also compile these functions separately and load with the lexical analyzer.
- c. Example :

```
int yywrap(){  
    int main(){  
        yylex();  
        //Actions at end of input  
        return 0;  
    }  
}
```

Important functions

- **yyparse()**
 - it parses (i.e builds the parse tree) of lexeme .*
 - returns a value of 0 if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is incorrect and error recovery is impossible.
 - does not do its own lexical analysis. In other words, it does not pull the input apart into tokens ready for parsing. Instead, it calls a routine called **yylex()** everytime it wants to obtain a token from the input.
- **yylex()**
 - Implies the main entry point for lex, reads the input stream generates tokens, returns zero at the end of input stream .
 - It is called to invoke the lexer (or scanner) and each time yylex() is called, the scanner continues processing the input from where it last left off.

-
- It returns a value indicating the *type* of token that has been obtained. If the token has an actual *value*, this value (or some representation of the value, for example, a pointer to a string containing the value) is returned in an external variable named **yylval**.
 - **yylval**
 - Contains the token value .
 - **yyval**
 - A local variable .*
 - **yywrap()**
 - it is called by lex when input is exhausted (or at EOF). default yywrap always return 1.
 - **yyless(k)**
 - returns the first k characters in **yytext** .
 - **yytext**
 - A buffer that holds the input characters that actually match the pattern (i.e lexeme) or say a pointer to the matched string .
 - **yylen**
 - The length of the lexeme .
 - **yyin**
 - The input stream pointer (i.e it points to an input file which is to be scanned or tokenized), however the default input of default main() is stdin .
 - **yyout**
 - The output stream pointer (i.e it points to a file where it has to keep the output), however the default output of default main() is stdout .
 - **yymore()**
 - Returns the next token .

Steps

Installation

Ubuntu

```
sudo apt-get update  
sudo apt-get install flex
```

Windows

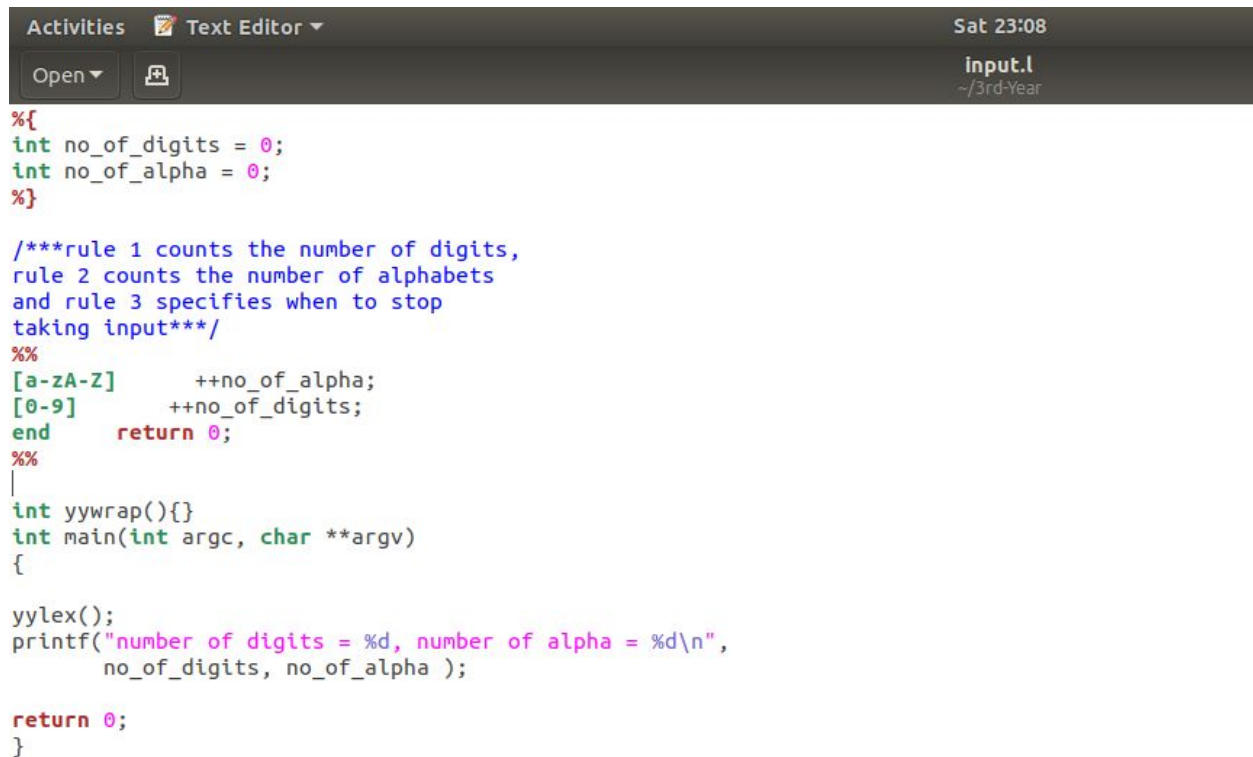
From [Flex for Windows](#)

Run a program on ubuntu:

1. Write program in file then save it with *.l or *.lex format.

Example :

Program that counts the number of digits and number of alphabets in a string.



```
%{
int no_of_digits = 0;
int no_of_alpha = 0;
}%

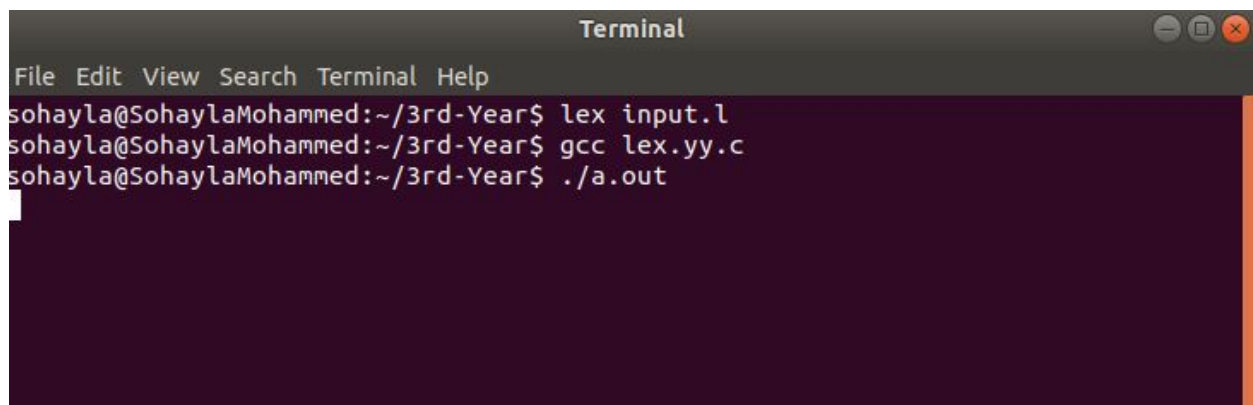
/**rule 1 counts the number of digits,
rule 2 counts the number of alphabets
and rule 3 specifies when to stop
taking input**/
%%
[a-zA-Z]      ++no_of_alpha;
[0-9]         ++no_of_digits;
end          return 0;
%%
|
int yywrap(){}
int main(int argc, char **argv)
{

yylex();
printf("number of digits = %d, number of alpha = %d\n",
      no_of_digits, no_of_alpha );

return 0;
}
```

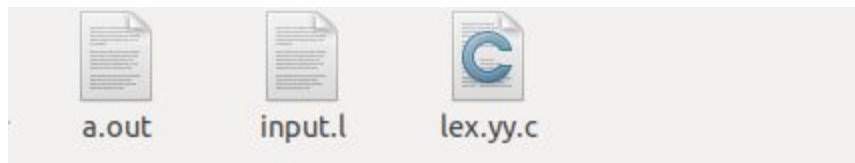
2. Open terminal then go to saved file directory using 'cd' and run the following commands.

- "lex file_name.l" or "lex file_name.lex" depending on the extension that the file is saved with.
- "gcc lex.yy.c"
- "./a.out"



```
Terminal
File Edit View Search Terminal Help
sohayla@SohaylaMohammed:~/3rd-Year$ lex input.l
sohayla@SohaylaMohammed:~/3rd-Year$ gcc lex.yy.c
sohayla@SohaylaMohammed:~/3rd-Year$ ./a.out
```

The directory should look like this afterwards :



3. Start entering your input in terminal. To end the process either press CTRL+D or use some rule in the “Rules” section.

I used rule that when end is entered the process terminates.

Example :


A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
sohayla@SohaylaMohammed:~/3rd-Year$ lex input.l
sohayla@SohaylaMohammed:~/3rd-Year$ gcc lex.yy.c
sohayla@SohaylaMohammed:~/3rd-Year$ ./a.out
Ab01Cde2345FgHIjk678880

end
number of digits = 12, number of alpha = 11
sohayla@SohaylaMohammed:~/3rd-Year$
```


For the given language

Input.l file

```
Open ▾  input.l ~/3rd-Year
input.l x

%{
%}
letter [a-zA-Z]
digit [0-9]

%%
"if"|"else"|"while"|"boolean"|"float"|"int" {printf( "%s \n", yytext);}
{letter}({letter}|{digit})* {printf( "id \n");}
{digit}+|{digit}+"."{digit}+{"E"{digit}+}? {printf( "num \n");}
"=="|"!="|">"|>="|"<"|<=" {printf( "relop \n");}
"=" {printf( "assign \n"); }
";"|"|"("|")"|"{"|"}" {printf( "%s \n", yytext);}
"+"|"-" {printf( "addop \n");}
"*"|"/" {printf( "mulop \n");}
"{"[^}\n]*"}
[ \t\n]+
.          printf( "Unrecognized character: %s\n", yytext );
%%
FILE *yyin;

int yywrap(){}
int main(int argc, char **argv) {
    yyin = fopen( "java.txt", "r" );
    yylex();
    fclose(yyin);

return 0;
}
```

Java.txt file

```
Open ▾  java.txt ~/3rd-Year
java.txt x

int sum , count , pass ,
mnt; while (pass != 10)
{
pass = pass + 1 ;
}
```

Output

```
Terminal
File Edit View Search Terminal Help
sohayla@SohaylaMohammed:~/3rd-Year$ lex input.l
sohayla@SohaylaMohammed:~/3rd-Year$ gcc lex.yy.c
sohayla@SohaylaMohammed:~/3rd-Year$ ./a.out
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
sohayla@SohaylaMohammed:~/3rd-Year$
```