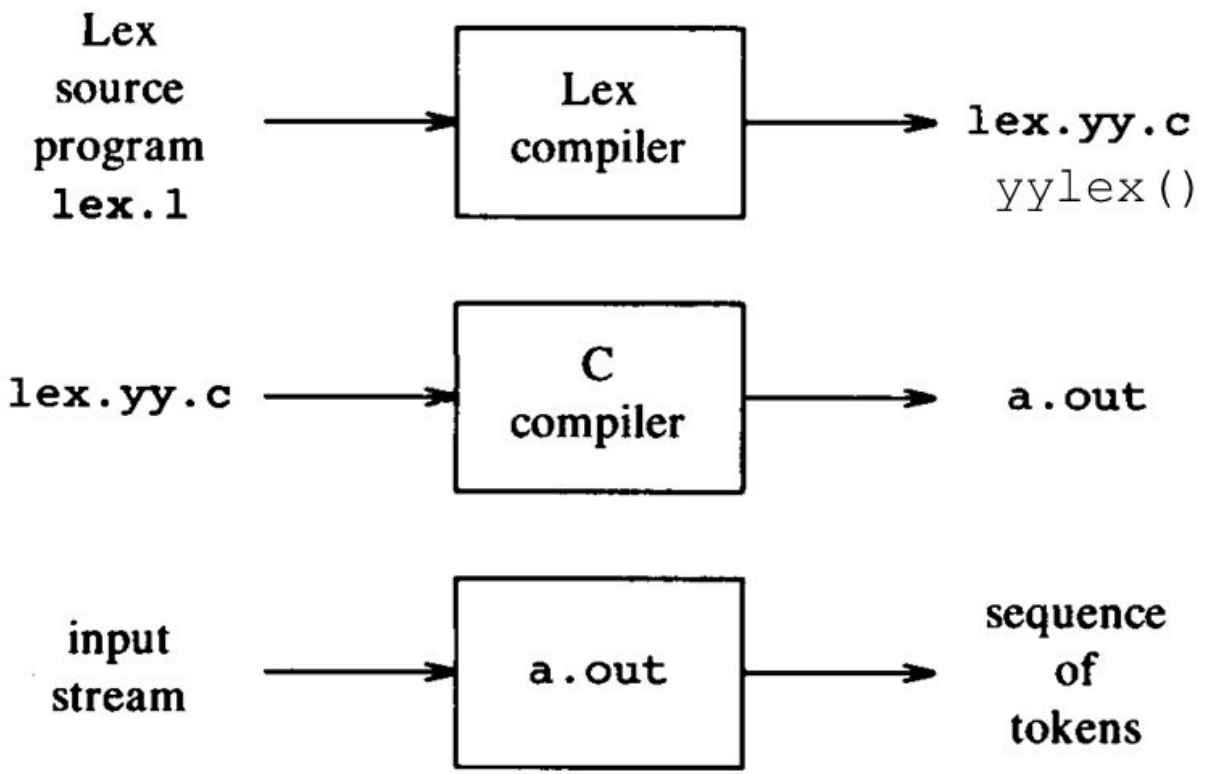
FLEX-Tool

- It is a tool used for generating lexical analyzer.
- It scans the input stream and divide it into units such as tokens.
- It uses regular expression to match and tokenize the input stream.
- The regular expressions are provided by the users in the source specification given to lex.
- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.

Working


Lex
source
program
lex.l
Lex
compiler
lex.yy.c
yylex()
lex.yy.c
C
compiler
a.out
input
stream
a.out
sequence
of
tokens

LEX Program Structure

Divided into 3 sections:

1. **Definition Section:** The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{ %}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c
2. **Rules Section:** 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. The rule section is enclosed in **"%% %%"**.
3. **User Code Section:** This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Program Structure

```
%{
  Declaration/Definitions
%}

%%
   pattern  action
%%

   User code section
```

[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘, ‘ or 9
[0 9]	either 0, ‘ ‘ or 9
[-09]	either -, 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a’s i.e, aaaa
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x | y)z	wxz or wyz

Program to count number of tokens in the input line

```
/*** One variable in definition section
to count number of tokens ***/
%{
int n= 0;
%}
```

```
// Rule Section has rules to classify tokens
%%
[a-zA-Z_][a-zA-Z0-9_]*|”<=“|”==“|”=“|”+”|”-”|”*”|[(){}|,;]|[0-9]*”.”[0-9]+|[0-9]+  {n++;}
%%
```

```
// Code Section
int yywrap(){}
int main(){
yylex();   //main flex function that runs the rules section each time it is called the scanner continues processing the input stream.
printf("\n total number of token present in the source code: %d", n);
}
```

Lex predefined variables

- yytext: a string containing the lexeme.
- yyleng: the length of the lexeme.
- yyin: the input stream pointer.
- yyout: the output stream pointer.

Program to print and count number, name and length of each token.

```
%{
int n= 0;
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*|”<=“|”==“|”=“|”+”|”-”|”*”|[(){}|,;]|[0-9]*”.”[0-9]+|[0-9]+  {n++; printf(“%s = %d\n”, yytext, yyleng);} //scanner stores text that matches an expression in a character array called yytext[]
%%
int yywrap(){}
int main(){
yylex();   //main flex function that runs the rules section
printf(“\n total number of token present in the source code: %d”, n);
}
```

Execution

- Save the program with extension .l
- On command prompt

  > flex filename.l

  > gcc lex.yy.c

  > ./a.out

  > Provide input if required