Lex Programs

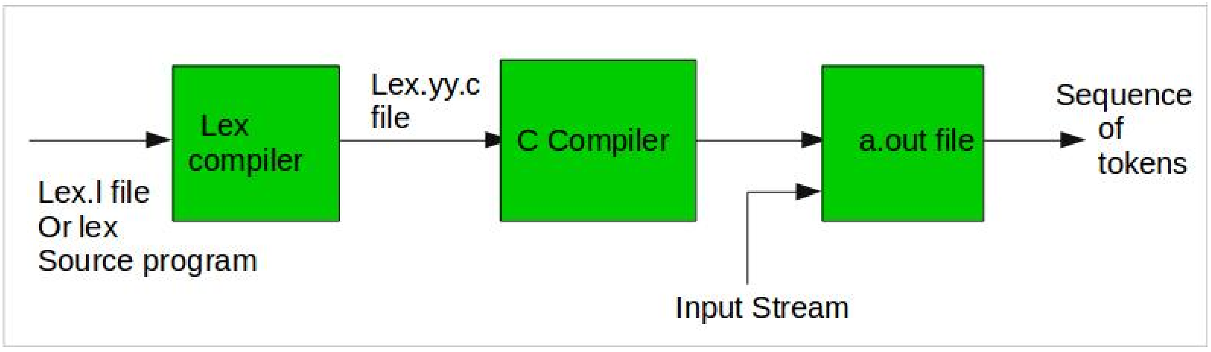
FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) .It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code. Bison produces 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** .

The function yylex() is the main flex function that runs the Rule Section and extension (.l) is the extension used to save the programs.

Installing Flex on Ubuntu:

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

How flex is used



Step 1: An input file describes the lexical analyzer to be generated named lex.l is written in lex language. The lex compiler transforms lex.l to C program, in a file that is always named lex.yy.c.
Step 2: The C compiler compile lex.yy.c file into an executable file called a.out.
Step 3: The output file a.out take a stream of input characters and produce a stream of tokens.

Program Structure:

In the input file, there are 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**

Syntax:

```
%{

   // Definitions

%}
```

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 "%% %%".

Syntax:

```
%%

pattern  action

%%
```

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.

Basic Program Structure:

```
%{

// Definitions

%}

%%

Rules

%%


User code section
```

How to run the program:

To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.

Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with

Step 2: gcc lex.yy.c

Step 3: ./a.out

Step 4: Provide the input to program in case it is required

Note: Press Ctrl+D or use some rule to stop taking inputs from the user.

Exercises

Lex program to count the number of words

```
/*lex program to count number of words*/

%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}

/* Rules Section*/

%%
([a-zA-Z0-9])*    {i++;} /* Rule for counting
                          number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%

int yywrap(void){}

int main()
{
   // The function that starts the analysis
   yylex();
return 0;
}
```

Implementation of Lexical Analyser using LEX tool

AIM

Implementation a lexical analyzer for given language using LEX tool

ALGORITHM

Token declaration Part

1. Declare token named Head as #include”<”.*”>”
2. Declare token named Key as “int”|”float”|”char”|”while”|”if”|”double”|”for”|”do”|”goto”|”void”
3. Declare token named Digit as [0-9]+
4. Declare token named Id as [a-z][A-Z0-9a-z0-9]*|_[a-z][A-Z0-9a-z0-9]*
5. Declare token named Op as “+”|”-“|”*”|”/”

Translation Rule Part

1. If input lexeme matches with token {Head} then print it is a header file.
2. If input lexeme matches with token {Key} then print it is a keyword.
3. If input lexeme matches with token {Digit} then print it is a Digit.
4. If input lexeme matches with token {Id} then print it is an Identifier.
5. If input lexeme matches with token {Op} then print it is an operator.
6. If input lexeme matches with token “.”, then skip other patterns.

Auxiliary Procedure Part

1. Start
2. Read the input file.
3. If input symbol is a keyword, print it is a keyword.
4. Check whether the current input is #include “<”.*”>”, then print it is a header file.
5. Check whether the current input is digit, then print it is a digit.
6. Check whether the current input contains any identifier, then print it is an identifier.
7. Check whether the current input contains any operator, then print it is an operator.
8. Repeat the steps 2-8 until EOF.
9. Stop.

PROGRAM

```
%{
#include<stdio.h>
#include<string.h>
```

```
char key[100][100],head[100][100],dig[100][100],op[100][100],id[100][100];
int i=0,j=0,k=0,l=0,a=0,b=0,c=0,d=0,m=0,n=0;
%}

KW "int"|"while"|"if"|"else"|"for"|"char"|"float"|"case"|"switch"
HF "#include<".*">"
OP "+"|"-"|"*"|"/"|"="
DIG [0-9]*|[0-9]*"."[0-9]+
ID [a-zA-Z][a-zA-Z0-9]*

%%
{KW} {strcpy(key[i],yytext);i++;}
{HF} {strcpy(head[j],yytext);j++;}
{DIG} {strcpy(dig[k],yytext);k++;}
{OP} {strcpy(op[m],yytext);m++;}
{ID} {strcpy(id[n],yytext);n++;}
. {}
%%

main()
{
	yyin=fopen("input.c","r+");
	yylex();
	printf("\nThe keywords are");
	for(a=0;a<i;a++)
	{
		printf("\n%s",key[a]);
	}
	printf("\nThe headerfiles are ");
	for (b=0;b<j;b++)
	{
		printf("\n%s",head[b]);
	}

	printf("\nThe digits are");
	for(c=0;c<k;c++)
	{
		printf("\n%s",dig[c]);
	}
	printf("\noperators ...");
	for (d=0;d<m;d++)
	{
		printf("\n%s",op[d]);
	}
	printf("\nidentifiers....");
	for(d=0;d<n;d++)
	{
		printf("\n%s",id[d]);
```

```
        }
}

int yywrap()
{
        printf("Errors..\n");
        return 1;
}
```