Compiler Construction Experiment 02

Shubham Golwal | 2020300015 | TE COMPS
16-02-23

AIM: To perform lexical analysis.

THEORY:

What is Lexical Analysis?
Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.
Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. The role of Lexical Analyzer in compiler design is to read character streams from the source code, check for legal tokens, and pass the data to the syntax analyzer when it demands.

What's a lexeme?
A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.
What's a token?
Tokens in compiler design are the sequence of characters which represents a unit of information in the source program.

What is Pattern?
A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

Lexical Analyzer Architecture:
The main task of lexical analysis is to read input characters in the code and produce tokens. Lexical analyzer scans the entire source code of the program. It identifies each token one by one. Scanners are usually implemented to produce tokens only when requested by a parser. Here is how recognition of tokens in compiler design works-

1. "Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parse

CODE:

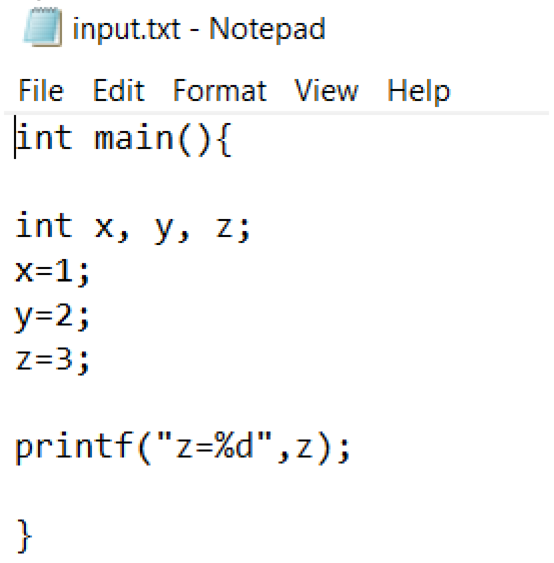
Given code:

```
%{
#include<stdio.h>
%}
%%
if|else|while    { printf("Keyword\n");}
([a-z]([a-z0-9])*)  { printf("Identifier\n");}
[0-9]+          { printf("Number\n");}
%%
int main()
{
yylex();
return 0;
}
int yywrap()
{
}
```

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

%option noyywrap

%%

#.   {total++; fprintf(yyout," Preprocessor Directive   : %s\n\n",yytext);}

[''|,|;(|)|{|}|.|_] {total++; fprintf(yyout," Delimiters    : %s\n\n",yytext);}

[[]]    {total++; fprintf(yyout," Delimiters    : %s\n\n",yytext);}

[-+*/%] {total++; fprintf(yyout," Arithmetic Operator   : %s\n\n",yytext);}

([<>]=?)|([!=]=)    {total++; fprintf(yyout," Relational Operator   :
%s\n\n",yytext);}

(&&)|(\|\|)|(!) {total++; fprintf(yyout," Logical Operator  : %s\n\n",yytext);}
```

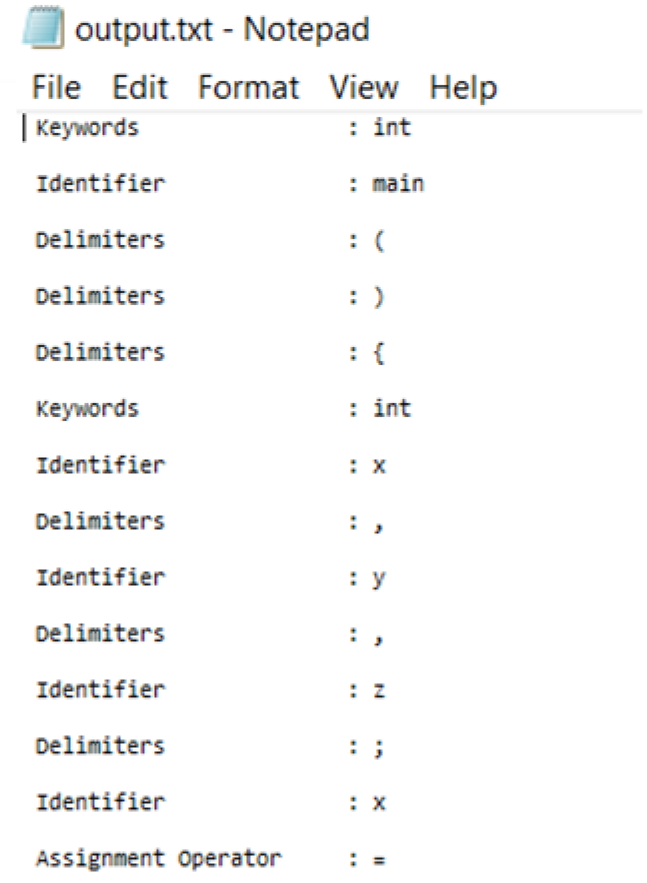
```
("if")|("else")|("switch")|("case")|("default")|("break")|("int")|("float")|("cha
r")|("double")|("long")|("for")|
("while")|("do")|("void")|("goto")|("auto")|("signed")|("const")|("extern")|("reg
ister")|("unsigned")|("return")|
("continue")|("enum")|("sizeof")|("struct")|("typedef ")|("union")|("volatile")
{total++; fprintf(yyout," Keywords  : %s\n\n",yytext);}

"printf"    {total++;   fprintf(yyout," Standadrd IO    :   %s\n\n",yytext);}

"//".*  {total++;   fprintf(yyout," Single line comment :   %s\n\n",yytext);}

"/*"([^\*]*|[\*]*)*"*/" {total++;   fprintf(yyout," Multi line
comment  :   %s\n\n",yytext);}

[a-zA-Z_][a-zA-Z0-9_]*  {total++;   fprintf(yyout,"
Identifier  :   %s\n\n",yytext);}

[0-9]*"."[0-9]+ {total++;   fprintf(yyout," Floating point
value    :   %s\n\n",yytext);}

[-][0-9]*"."[0-9]+  {total++;   fprintf(yyout," Negative Floating point
:   %s\n\n",yytext);}

[0-9]+  {total++;   fprintf(yyout," Integer :   %s\n\n",yytext);}
"-"[0-9]+   {total++;   fprintf(yyout," Negative Integer    :   %s\n\n",yytext);}

=   {total++;   fprintf(yyout," Assignment Operator :   %s\n\n",yytext);}

["]([^"\\\n]|\\.|\\\n)*["]  {total++;   fprintf(yyout,"

String  :   %s\n\n",yytext);}

[ \t\n"]+   ;
%%

int main()
{
extern FILE *yyin, *yyout;
yyin = fopen("input.txt","r");
yyout = fopen("output.txt","w"); yylex();
fprintf(yyout,"\n\nTotal Tokens = %d", total);
return 0;
}
```

Input:



```
int main(){

int x, y, z;
x=1;
y=2;
z=3;

printf("z=%d",z);

}
```

Output:



```
Keywords                : int

Identifier              : main

Delimiters              : (

Delimiters              : )

Delimiters              : {

Keywords                : int

Identifier              : x

Delimiters              : ,

Identifier              : y

Delimiters              : ,

Identifier              : z

Delimiters              : ;

Identifier              : x

Assignment Operator     : =
```

CONCLUSION:

- In this experiment, we learnt that we can create a lex tool which identifies and classifies the given tokens like variables, numbers, special symbols, operators for a given language.
- Learnt that we can create a Lexical tool for any language, but the result output file is always in the C language.
- Learnt that the Lexical Analyzer creates a C file which contains all the data of .l file in C language, which when compiled, gives an a.out file
- To execute we have to also run: lex.yy.c
- Also came to know that we can run ./a.out < filename to give input to the command from the file

REF.:

1. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
2. https://blog.devgenius.io/a-simple-lexical-analyser-aaa4329b18d0