**Practical No-2(b)**

**Aim :**  Using Flex or any scanner generator tool, generate scanner for a language.

**Theory :**

* Generate scanner using lex , which can identify following language –“D” ,
* display appropriate messages on recognizing them

**Rules of language “D”**

1. Keywords—if, for ,while, do, exit, else, case, switch, until
2. Identifiers-starting with alphabet(big or small) and followed by one or more alphabets or

numbers

1. Numbers-
2. float number 🡪having one ‘.’ and valid formats are 🡪 . Num or Num.Num or Num . 0
3. integer number🡪simple number, starting with digit followed by zero or more digits
4. Single line comments starting with \*\*\*
5. White space characters are - blanks and tabs
6. Punctuations are –[ ,],{,},(,) and comma
7. Operators :=,<>,= = ,!= , <=, <=, (relational),+,-,\*,% (mathametical)
8. String constant in ‘ ‘ (Note-no enclosing quotes allowed in language)

**OR**

Construct a lexical analyzer which

* Allow white spaces, numbers and arithmetic operators in an expression
* Return tokens and attributes to the syntax analyzer
* A global variable tokenval is set to the value of the number
* Design requires that
  + A finite set of tokens be defined
  + Describe strings belonging to each token

We now try to construct a lexical analyzer for a language in which white spaces, numbers and arithmetic operators in an expression are allowed. From the input stream, the lexical analyzer recognizes the tokens and their corresponding attributes and returns them to the syntax analyzer.

To achieve this, the function returns the corresponding token for the lexeme and sets a global variable, say tokenval , to the value of that token. Thus, we must define a finite set of tokens and specify the strings belonging to each token.

To handle keywords, we consider the keywords themselves as lexemes. We store all the entries corresponding to keywords in the symbol table while initializing it and do lookup whenever we see a new lexeme. Now, whenever a lookup is done, if a nonzero value is returned, it means that there already exists a corresponding entry in the Symbol Table. So, if someone tries to use a keyword as an identifier, it will not be allowed as an identifier with this name already exists in the Symbol Table. For instance, consider the tokens DIV and MOD with lexemes "div" and "mod". We initialize symbol table with insert("div", DIV) and insert("mod", MOD). Any subsequent lookup now would return a nonzero value, and therefore, neither "div" nor "mod" can be used as an identifier.

**Token:** A token is a syntactic category. Sentences consist of a string of tokens. For example number, identifier, keyword, string etc are tokens.

**Lexeme:** Sequence of characters in a token is a lexeme. For example 100.01, counter, const, "How are you?" etc are lexemes.

**Pattern:** Rule of description is a pattern. For example letter (letter | digit)\* is a pattern to symbolize a set of strings which consist of a letter followed by a letter or digit. In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. This pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token

**Symbol Table**

* Stores information for subsequent phases
* Interface to the symbol table
  + Insert(s,t): save lexeme s and token t and return pointer
  + Lookup(s): return index of entry for lexeme s or 0 if s is not found

Implementation of symbol table

* Fixed amount of space to store lexemes. Not advisable as it waste space.
* Store lexemes in a separate array. Each lexeme is separated by eos. Symbol table has pointers to lexemes.

**How to handle keywords?**

* Consider token DIV and MOD with lexemes div and mod.
* Initialize symbol table with insert( "div" , DIV ) and insert( "mod" , MOD).
* Any subsequent lookup returns a nonzero value, therefore, cannot be used as an identifier .

**How to describe tokens?**

* Programming language tokens can be described by regular languages
* Regular languages
  + Are easy to understand
  + There is a well understood and useful theory
  + They have efficient implementation

**Program:**

%{

#include<stdio.h>

int keyword=0,Identifier=0,number=0,Spec=0,op=0,comm=0,dec=0,wsp=0,hd=0;

%}

%%

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"regiter"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"void"|"volatile"|"while" {printf("Keyword : %s\n",yytext);keyword++;}

[a-z|A-Z][a-z|0-9]\* {printf("Identifier : %s\n",yytext);Identifier++;}

[0-9]+ {printf("Number : %s\n",yytext);number++;}

","|"<"|">"|"."|"\_"|")"|";"|"("|"$"|":"|"%"|"]"|"["|"#"|"?"|"&"|"}"|"{"|"^"|"!"|"~"|"\\"|"\'"|"\"" {printf("Spec symbol :%s\n",yytext);Spec++;}

"+"|"\*"|"-"|"/"|"=" {printf("Operator : %s\n",yytext);op++;}

[/]+[/]\*+[a-z|A-Z|0-9|" "]+ {printf("Comment : %s\n",yytext);comm++;}

[0-9]+.[0-9]+ {printf("Float : %s\n",yytext);dec++;}

[" "" "] {wsp++;}

[#include<a-z.h>]\* {printf("Header file : %s\n",yytext);hd++;}

%%

int yywrap()

{

return 1;

}

main()

{

FILE \*fp=fopen("ggccp2b.txt","r");

if(!fp)

printf("Couldn't open file");

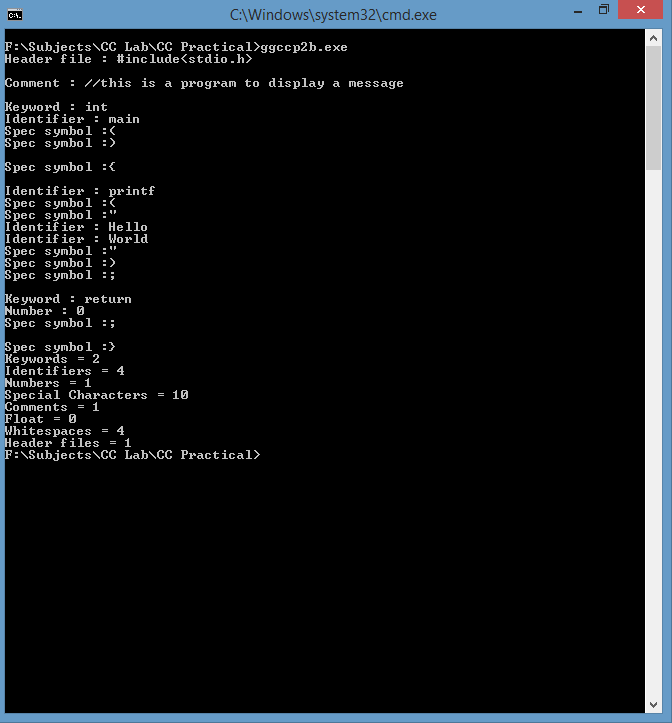
yyin=fp;

yylex();

printf("Keywords = %d\nIdentifiers = %d\nNumbers = %d\nSpecial Characters = %d\nComments = %d\nFloat = %d\nWhitespaces = %d\nHeader files = %d",keyword,Identifier,number,Spec,comm,dec,wsp,hd);

}

**Output:**

****

**Conclusion:**

A basic language scanner for C language has been constructed using FLEX. Comments, identifiers, keywords, whitespaces, special symbols, simple and float numbers, special characters and header files have been identified with the language scanner.