Objectives:

1/ To learn how to tokenize keywords of a programming language.

2/ To tokenize c program variable.

Introduction:

Lexical analysis is a fundamental phase in the compilation process. It involves the transformation of source code into a sequence of tokens, which are the smallest units of meaning in a programming language. It has used fixed rules and regular expressions to tokenize source code. Here we see how to ~~to~~ identify variable declaration and count of variables and statements.

Code:

```
%{
# include <stdio.h>
# include <string.h>
Char* datatype={"int", "float", "double","char", NULL};
int isDataType (char* word)
{
   for (int i=0 ;datatype[i] != NULL; i++)
   {
      if (strcmp (datatype[i], word) ==0)
      {   return 1;
      }
   }
   return 0;
}
int var=0, stat =0;
%}

DataType "int"|"float"| "double"| "char"
variable [a-zA-Z_][a-zA-Z0-9_]*
%%
{DataType}[ ]+{variable}([,][ ]*{variable})* {
   Char* token = strtok(yytext, "  ,\t");
   char* type;
```

```
  while (token != NULL)
  {
     if (!isDatatype(token)) {
         var++;
         printf ("variable");
     }
     else {
          type = token;
     }
     token= strtok (NULL, "   ,\t");
  }
}
";"  {stat++;}
%%
int main()
{
  yyin = fopen ("input.txt", "r");
  yylex();

  printf ("variables = %d \n", var);
  printf ("statements = %d\n", stat);
}
```

<u>input</u>

```
int a, b, c;
```

<u>Output</u>

```
variables = 3
statements = 1
```

Discussion:

We faced troubles in writing regex for variable declaration. specially for the variables separated by (,) ~~are~~ seems difficult to write regex. As we are newbie to lexical analysis, we need more practise. However, we managed to implement it using flex.

Conclusion:

Lexical analysis is a fundamental phase in compiler design. If we can correctly tokenize all variables, keywords, datatypes and so on, It will help us to continue further processes of compiler design.

References:

1/ Lab resource