29/01/2021

A1: Implementation of Lexical Analyzer for the patterns - (identifier, comments, operators, constants)

185001188
Vanathi G
CSE-C

Aim - Develop a Lexical analyzer using C to recognize the patterns namely - identifiers, constants, comments and operators - using the given regular expressions.

Program -

```
/* PROGRAM : Implementation of Lexical Analyzer for the patterns (identifier, comments,
operators, constants) */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define DELIMITER (c == ' ' || c == '\n' || c == ';')
#define MAX 32

typedef struct
{
  char name[MAX];
  int n;

}identifier;

int isKeyword(identifier id);

void main()
{
  char c, prev;
  int err_flag = 0, digit_seen = 0, char_seen = 0;

  // 0 for start state; revert to this state after encountering the delimiter
  int state = 0;

  identifier id;
  strcpy(id.name, "");
  id.n = 0;

  // Keep reading characters until EOF
  while((c = getchar()) != EOF)
  {
    // For every state, define what state to move to based on read character
    switch(state)
    {
```

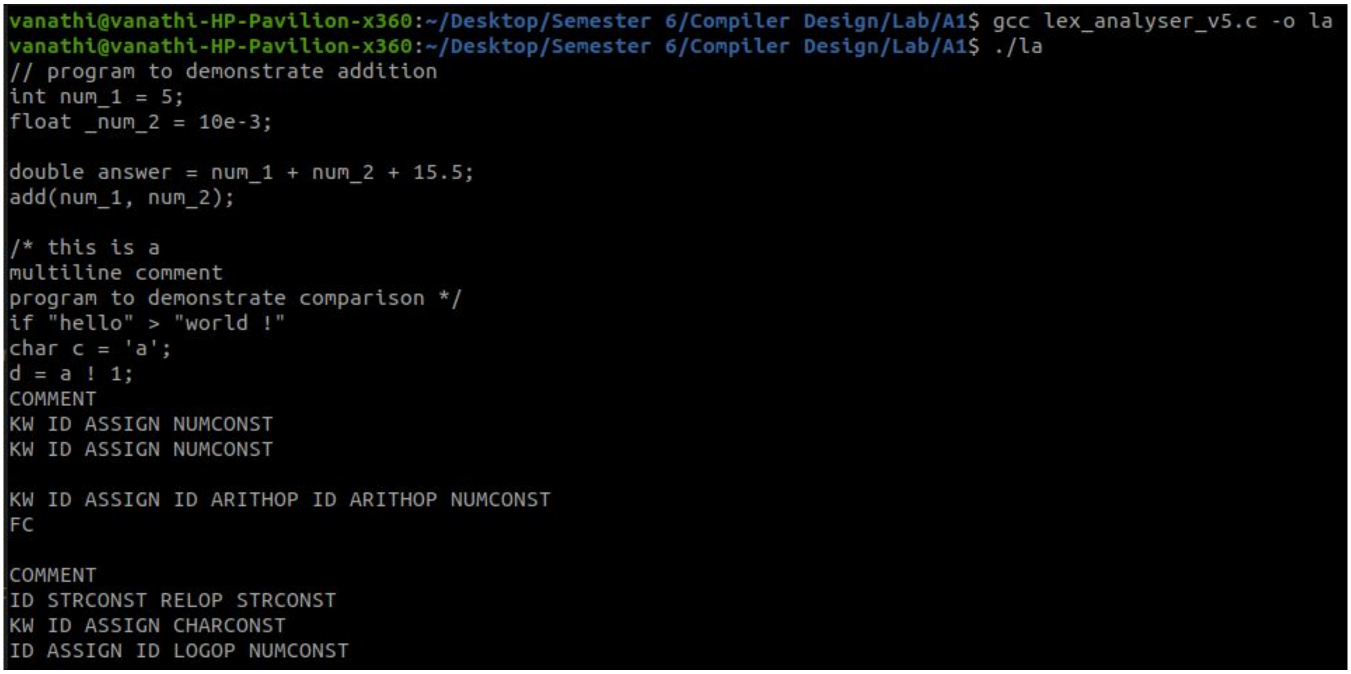
```
case 0:
{
  if(c == '<' || c == '>')
    state = 1;
  else if(c == '=')
    state = 2;
  else if(c == '!')
    state = 4;
  else if(c == '+' || c == '-' || c == '*' || c == '%')
    state = 5;
  else if(isdigit(c))
    state = 6;
  else if(c == '\'')
    state = 10;
  else if(c == '"')
    state = 11;
  else if(isalpha(c) || c == '_')
  {
    id.name[id.n++] = c;
    state = 14;
  }
  else if(c == '/')
    state = 15;
  else if(c == '\n')
    c = '\n';
  else
    err_flag = 1;
  break;
}
case 1:
{
  if DELIMITER
  {
    printf("RELOP ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
    err_flag = 1;
  break;
}
case 2:
{
  if DELIMITER
  {
    printf("ASSIGN ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
    err_flag = 1;
  break;
}
case 3:
{
  if DELIMITER
  {
    printf("RELOP ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 4:
{
  if DELIMITER
  {
    printf("LOGOP ");
    state = 0;
  }
  else if(c == '=')
    state = 3;
  else
    err_flag = 1;
  break;
}
case 5:
{
  if DELIMITER
  {
    printf("ARITHOP ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 6:
{
  if DELIMITER
  {
    printf("NUMCONST ");
    state = 0;
  }
  else if(isdigit(c))
    state = 6;
  else if(c == '.')
    state = 7;
  else if(c == 'E' || c == 'e')
```

```
    state = 8;
  else
    err_flag = 1;
  break;
}
case 7:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(isdigit(c))
  {
    digit_seen = 1;
    state = 7;
  }
  else
    err_flag = 1;
  break;
}
case 8:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(c == '+' || c == '-')
    state = 9;
  else if(isdigit(c))
  {
    digit_seen = 1;
    state = 8;
  }
  else
    err_flag = 1;
  break;
}
case 9:
{
  if(DELIMITER && digit_seen == 1)
  {
    printf("NUMCONST ");
    state = 0;
    digit_seen = 0;
  }
  else if(isdigit(c))
  {
    digit_seen = 1;
    state = 9;
  }
  else
    err_flag = 1;
  break;

}
case 10:
{
  if(c == '\'')
  {
    state = 12;
    char_seen = 0;
  }
  else if(c!='\n' && char_seen == 0)
  {
    char_seen = 1;
    state = 10;
  }
  else
    err_flag = 1;
  break;
}
case 11:
{
  if(c == '"')
    state = 13;
  else if(c != '\n')
    state = 11;
  else
    err_flag = 1;
  break;
}
case 12:
{
  if DELIMITER
  {
    printf("CHARCONST ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 13:
{
  if DELIMITER
  {
    printf("STRCONST ");
    state = 0;
  }
  else
    err_flag = 1;
  break;
}
case 14:
{
  if DELIMITER
  {
    id.name[id.n]='\0';
    if(isKeyword(id))
      printf("KW ");
    else
      printf("ID ");
    state = 0;
    strcpy(id.name, "");
    id.n = 0;
  }
  else if(isalnum(c) || c == '_')
  {
    id.name[id.n++] = c;
    state = 14;
  }
  else if(c == '(')
    state = 19;
  else
    err_flag = 1;
  break;
}
case 15:
{
  if DELIMITER
  {
    printf("ARITHOP ");
    state = 0;
  }
  else if(c == '/')
    state = 16;
  else if(c == '*')
    state = 17;
  else
    err_flag = 1;
  break;
}
case 16:
{
  if(c == '\n')
  {
    printf("COMMENT ");
    state = 0;
  }
```

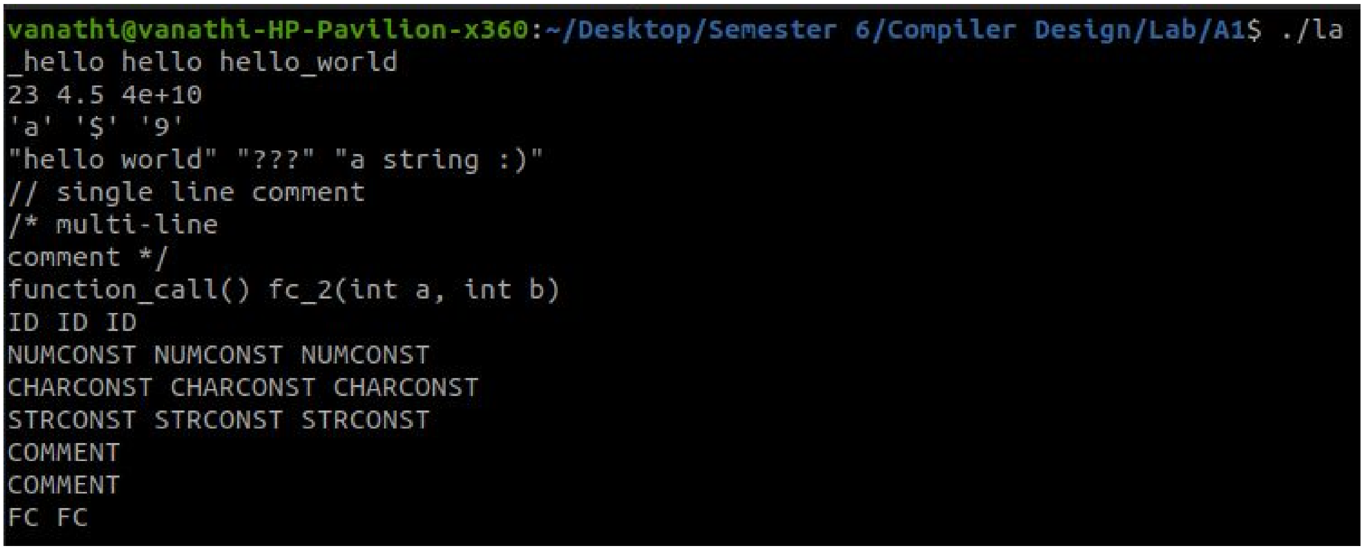
```
    else
      state = 16;
    break;
  }
  case 17:
  {
    if(c == '*')
      state = 18;
    else
      state = 17;
    break;
  }
  case 18:
  {
    if(c == '/')
    {
      printf("COMMENT ");
      state = 0;
    }
    else
      state = 17;
    break;
  }
  case 19:
  {
    if(c == ')')
      state = 20;
    else if(isalnum(c) || c == ' ' || c == '_' || c == ',')
      state = 19;
    else
      err_flag = 1;
    break;
  }
  case 20:
  {
    if DELIMITER
    {
      printf("FC ");
      state = 0;
    }
    else
      err_flag = 1;
    break;
  }

}
if(err_flag == 1)
{
  printf("Invalid token!\n");
  break;
}
```

```
    if(state != 17 && c == '\n')
      printf("\n");
  }
  printf("\n");

}

// Function to check whether it is an ID or keyword
int isKeyword(identifier id)
{
  char words[32][15] = {"auto", "break", "case", "char", "const", "continue", "default",
              "do", "double", "else", "enum", "extern", "float", "for", "goto",
              "if", "int", "long", "register", "return", "short", "signed", "sizeof",
              "static", "struct", "switch", "typedef", "union", "unsigned", "void",
              "volatile", "while"};
  for(int i=0; i<32; i++)
  {
    if(strcmp(id.name, words[i]) == 0)
      return 1;
  }
  return 0;
}
```

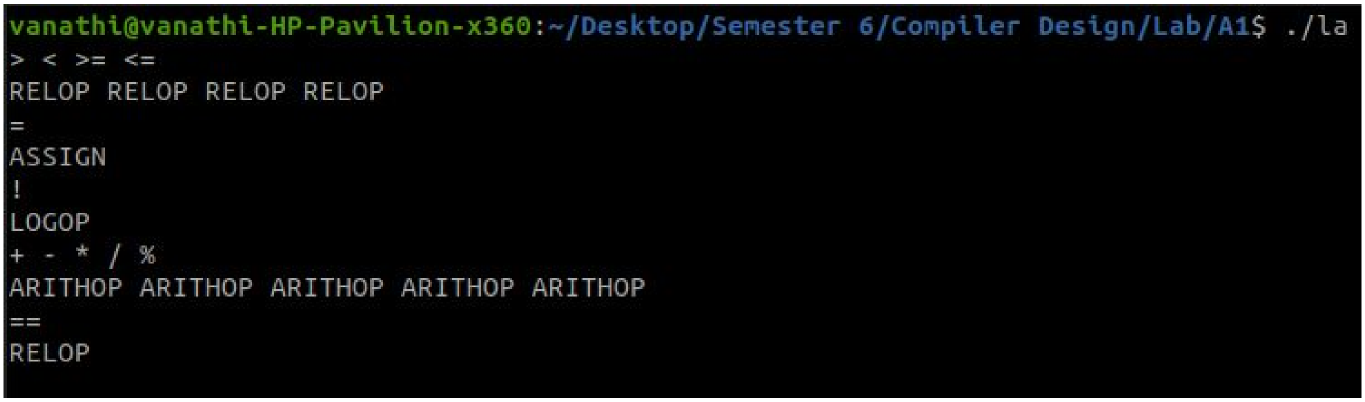
I/O Snapshots -



```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ gcc lex_analyser_v5.c -o la
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
// program to demonstrate addition
int num_1 = 5;
float _num_2 = 10e-3;

double answer = num_1 + num_2 + 15.5;
add(num_1, num_2);

/* this is a
multiline comment
program to demonstrate comparison */
if "hello" > "world !"
char c = 'a';
d = a ! 1;
COMMENT
KW ID ASSIGN NUMCONST
KW ID ASSIGN NUMCONST

KW ID ASSIGN ID ARITHOP ID ARITHOP NUMCONST
FC

COMMENT
ID STRCONST RELOP STRCONST
KW ID ASSIGN CHARCONST
ID ASSIGN ID LOGOP NUMCONST
```



```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
_hello hello hello_world
23 4.5 4e+10
'a' '$' '9'
"hello world" "???" "a string :)"
// single line comment
/* multi-line
comment */
function_call() fc_2(int a, int b)
ID ID ID
NUMCONST NUMCONST NUMCONST
CHARCONST CHARCONST CHARCONST
STRCONST STRCONST STRCONST
COMMENT
COMMENT
FC FC
```



```
vanathi@vanathi-HP-Pavilion-x360:~/Desktop/Semester 6/Compiler Design/Lab/A1$ ./la
> < >= <=
RELOP RELOP RELOP RELOP
=
ASSIGN
!
LOGOP
+ - * / %
ARITHOP ARITHOP ARITHOP ARITHOP ARITHOP
==
RELOP
```

Learning Outcomes -

1. I successfully developed a basic lexical analyser that recognizes various patterns such as constants, variables, operators, comments and function calls.
2. I learnt what a token is and why we need to convert the high-level language words to tokens before syntax analysis.
3. I understood the regular expressions used to represent these tokens and was also able to use the transition diagram to help implement the lexical analyser.
4. I found it challenging to implement the recognition of multi-line comments but I was successfully able to do it after constructing the required transitions in the transition diagram.
5. I also understood how a lexical analyser works and its functions in the compiler.