Week3

Name : NEERAJ KUMAR
Regno. : 220905536
Roll no. : 57
Title: CONSTRUCTION OF TOKEN GENERATOR

Sample Exercise:

1. **Write a program in C to identify the arithmetic and relational operators from the given input 'C' file.**

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

void main()
{
   char c, buf[10];
   FILE *fp = fopen("lab3_1.c", "r");
   c = fgetc(fp);
   if (fp == NULL)
   {
      printf("Cannot open file  \n");
      exit(0);
   }
   while (c != EOF)
   {
      int i = 0;
      buf[0] = '\0';
      if (c == '=')
      {
         buf[i++] = c;
         c = fgetc(fp);
         if (c == '=')
         {
            buf[i++] = c;
            buf[i] = '\0';
            printf("\n Relational operator : %s", buf);
         }
         else
         {
            buf[i] = '\0';
            printf("\n Assignment operator: %s", buf);
         }
      }
      else
      {
         if (c == '<' || c == '>' || c == '!')
         {
            buf[i++] = c;
            c = fgetc(fp);
```

```
            if (c == '=')
            {
               buf[i++] = c;
            }
            buf[i] = '\0';
            printf("\n Relational operator : %s", buf);
         }
         else
         {
            buf[i] = '\0';
         }
         c = fgetc(fp);
      }
   }
}
```

Output:

```
E:\labrelated\cdlab\week3>gcc sample.c
sample.c: In function 'main':

E:\labrelated\cdlab\week3>a.exe

 Relational operator : <
 Relational operator : >
 Relational operator : <
 Relational operator : >
 Relational operator : <
 Relational operator : >
 Relational operator : ==
 Assignment operator: =
 Relational operator : ==
```

Lab Exercise:

1) **Write functions to identify the following tokens.**
   a. Arithmetic, relational and logical operators.
   b. Special symbols, keywords, numerical constants, string literals and identifiers.

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

int isArithmeticOp(char *c)
{
   return (
      strcmp(c, "+") == 0 ||     strcmp(c, "-") == 0 || strcmp(c, "*") == 0 || strcmp(c, "/") == 0 ||
      strcmp(c, "++") == 0 ||       strcmp(c, "--") == 0 ||       strcmp(c, "%") == 0);
}

int isLogicalOp(char *c)
{
   return (      strcmp(c, "||") == 0 ||       strcmp(c, "&&") == 0 ||       strcmp(c, "!=") == 0);
}

int isRelationOp(char *c)
{
```

```
    return (       strcmp(c, "<") == 0 ||       strcmp(c, ">") == 0 ||       strcmp(c, "<=") == 0 ||
        strcmp(c, ">=") == 0 ||       strcmp(c, "=") == 0 ||       strcmp(c, "==") == 0);
}

int isSpecialSymbol(char *c)
{
    return (       strcmp(c, ";") == 0 ||       strcmp(c, ",") == 0 ||       strcmp(c, "(") == 0 ||
        strcmp(c, ")") == 0 ||       strcmp(c, "{") == 0 ||       strcmp(c, "}") == 0 ||
        strcmp(c, "[") == 0 ||       strcmp(c, "]") == 0 ||       strcmp(c, ".") == 0 ||
        strcmp(c, "&") == 0 ||       strcmp(c, "|") == 0 ||       strcmp(c, "^") == 0 ||
        strcmp(c, "~") == 0 ||       strcmp(c, "?") == 0 ||       strcmp(c, ":") == 0);
}

int isKeyword(char *c)
{
    char *keywords[] = {
        "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",
        "inline", "restrict", "bool", "complex", "imaginary"};

    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)
    {
        if (strcmp(c, keywords[i]) == 0)
        {
            return 1; // It's a keyword
        }
    }
    return 0; // Not a keyword
}

int isNumericalConstant(char *c)
{
    int hasDecimalPoint = 0, i = 0;
    // Check for optional sign
    if (c[i] == '+' || c[i] == '-')
    {
        i++;
    }
    // Check digits
    while (c[i] != '\0')
    {
        if (c[i] == '.')
        {
            if (hasDecimalPoint)
            {
                return 0; // More than one decimal point
            }
            hasDecimalPoint = 1; // Found a decimal point
        }
        else if (!isdigit(c[i]))
```

```
        {
            return 0;
        }
        i++;
    }
    return 1;
}

void identifyOperator(char *filename)
{
    char buffer[128];
    int i = 0, rowCount = 1;
    FILE *file1 = fopen(filename, "r");

    if (file1 == NULL)
    {
        printf("Error opening file: %s\n", filename);
        return;
    }

    char c;
    while ((c = fgetc(file1)) != EOF)
    {
        // Skip whitespace
        if (isspace(c))
        {
            if (c == '\n')
            {
                rowCount++;
            }
            continue;
        }

        // Handle token extraction for operators
        i = 0;
        if (ispunct(c))
        {
            // Handle single character operators
            buffer[i++] = c;
            // Check for two-character operators
            if (c == '+' || c == '-' || c == '!' || c == '<' || c == '>')
            {
                char next = fgetc(file1);
                if (next == c || next == '=' || next == '-')
                { // Handle ++, --, ==, !=, <=, >=
                    buffer[i++] = next;
                }
                else
                {
                    ungetc(next, file1); // Put back the character if it's not part of an operator
                }
            }
```

```
            buffer[i] = '\0';
            if (isArithmeticOp(buffer))
            {
               printf("\nArithmetic Operator: %s at row number: %d\n", buffer, rowCount);
            }
            else if (isRelationOp(buffer))
            {
               printf("\nRelational Operator: %s at row number: %d\n", buffer, rowCount);
            }
            else if (isLogicalOp(buffer))
            {
               printf("\nLogical Operator: %s at row number: %d\n", buffer, rowCount);
            }
            else if(isNumericalConstant(buffer))
            {
               printf("\nNumeric Constant: %s at row number: %d\n", buffer, rowCount);
            }
            else if(isSpecialSymbol(buffer))
            {
               printf("\nSpecial Symbol: %s at row number: %d\n", buffer, rowCount);
            }
            else if(isKeyword(buffer)){
               printf("\nKeyword is: %s at row number: %d\n", buffer, rowCount);
            }
         }
      }
      fclose(file1);
}

int main()
{
   char filename[128];
   printf("\nEnter the filename: ");
   scanf("%s", filename);
   identifyOperator(filename);
   return 0;
}
```

Output:

```
E:\labrelated\cdlab\week3>gcc lab3_1.c

E:\labrelated\cdlab\week3>a

Enter the filename: a.txt
Special Symbol: { at row number: 81

Arithmetic Operator: ++ at row number: 82

Special Symbol: ; at row number: 82

Special Symbol: } at row number: 83

Arithmetic Operator: / at row number: 84

Arithmetic Operator: / at row number: 84

Special Symbol: ( at row number: 85

Special Symbol: [ at row number: 85

Special Symbol: ] at row number: 85

Logical Operator: != at row number: 85

Special Symbol: ) at row number: 85

Special Symbol: { at row number: 86

Special Symbol: ( at row number: 87

Special Symbol: [ at row number: 87

Special Symbol: ] at row number: 87

Relational Operator: = at row number: 87

Relational Operator: = at row number: 87

Numeric Constant: . at row number: 87

Special Symbol: ) at row number: 87

Special Symbol: { at row number: 88
```

2) Design a lexical analyzer that includes a getNextToken() function for processing a simple C program. The analyzer should construct a token structure containing the row number, column number, and token type for each identified token. The getNextToken() function must ignore tokens located within single line or multi-line comments, as well as those found inside string literals. Additionally, it should strip out preprocessor directives.

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

```
#include <ctype.h>

#define max_token_LENGTH 100

typedef enum
{
   token_ID,
   token_NUMBER,
   token_OPERATOR,
   token_KEYWORD,
   token_STRING,
   token_COMMENT,
   token_PREPROCESSOR,
   token_UNKNOWN,
   token_EOF
} TokenType;

typedef struct
{
   int row;
   int col;
   TokenType type;
   char value[max_token_LENGTH];
} Token;

const char *keywords[] = {
   "int", "float", "double", "char", "return", "if", "else", "while", "for", "void", NULL};

int isKeyword(const char *word)
{
   for (int i = 0; keywords[i] != NULL; i++)
   {
      if (strcmp(word, keywords[i]) == 0)
      {
         return 1;
      }
   }
   return 0;
}

Token getNextToken(FILE *source, int *row, int *col)
{
   Token token;
   token.row = *row;
   token.col = *col;
   token.type = token_UNKNOWN;
   token.value[0] = '\0';

   int c;
   while ((c = fgetc(source)) != EOF)
   {
      (*col)++;
```

```
// Handle new lines
if (c == '\n')
{
   (*row)++;
   *col = 0;
   continue;
}

// Skip whitespace
if (isspace(c))
{
   continue;
}

// Handle comments
if (c == '/')
{
   c = fgetc(source);
   if (c == '/')
   { // Single-line comment
      while ((c = fgetc(source)) != EOF && c != '\n')
      {
         (*col)++;
      }
      (*row)++;
      *col = 0;
      continue;
   }
   else if (c == '*')
   { // Multi-line comment
      while (1)
      {
         c = fgetc(source);
         if (c == EOF)
            break;
         if (c == '*')
         {
            c = fgetc(source);
            if (c == '/')
            {
               break;
            }
         }
         if (c == '\n')
         {
            (*row)++;
            *col = 0;
         }
      }
      continue;
   }
```

```
    else
    {
        ungetc(c, source); // Not a comment, put back the character
        c = '/';
    }
}

// Handle preprocessor directives
if (c == '#')
{
    token.type = token_PREPROCESSOR;
    token.value[0] = c;
    int i = 1;
    while ((c = fgetc(source)) != EOF && c != '\n')
    {
        token.value[i++] = c;
        (*col)++;
    }
    token.value[i] = '\0';
    return token;
}

// Handle string literals
if (c == '"')
{
    token.type = token_STRING;
    int i = 0;
    token.value[i++] = c;
    while ((c = fgetc(source)) != EOF)
    {
        token.value[i++] = c;
        (*col)++;
        if (c == '"')
        {
            break;
        }
    }
    token.value[i] = '\0';
    return token;
}

// Handle identifiers and keywords
if (isalpha(c) || c == '_')
{
    token.type = token_ID;
    int i = 0;
    token.value[i++] = c;
    while (isalnum((c = fgetc(source))) || c == '_')
    {
        token.value[i++] = c;
        (*col)++;
    }
```

```
            ungetc(c, source);
            token.value[i] = '\0';
            if (isKeyword(token.value))
            {
               token.type = token_KEYWORD;
            }
            return token;
         }

         // Handle numbers
         if (isdigit(c))
         {
            token.type = token_NUMBER;
            int i = 0;
            token.value[i++] = c;
            while (isdigit((c = fgetc(source)))){
               token.value[i++] = c;
               (*col)++;
            }
            ungetc(c, source);
            token.value[i] = '\0';
            return token;
         }

         // Handle operators
         token.type = token_OPERATOR;
         token.value[0] = c;
         token.value[1] = '\0';
         return token;
      }

      token.type = token_EOF;
      return token;
   }

   int main()
   {
      char filename[128];
      printf("\nEnter the filename: ");
      scanf("%s", filename);

      FILE *file1 = fopen(filename, "r");
      if (file1 == NULL)
      {
         perror("Error opening file");
         return 1;
      }

      int row = 1, col = 0;
      Token token = getNextToken(file1, &row, &col);

      while (token.type != token_EOF)
```

```
    {
        printf("Row: %d, Col: %d, Type: %d, Value: %s\n", token.row, token.col, token.type,
token.value);
        token = getNextToken(file1, &row, &col);
    }

    fclose(file1);
    return 0;
}
```

Output:

```
E:\labrelated\cdlab\week3>gcc lab3_2.c

E:\labrelated\cdlab\week3>a.exe

Enter the filename: a.txt
Row: 1, Col: 0, Type: 6, Value: #include <stdio.h>
Row: 1, Col: 18, Type: 0, Value: VOID
Row: 2, Col: 4, Type: 0, Value: removeSpace
Row: 2, Col: 16, Type: 2, Value: (
Row: 2, Col: 17, Type: 0, Value: CHAR
Row: 2, Col: 21, Type: 2, Value: *
Row: 2, Col: 23, Type: 0, Value: srcFile
Row: 2, Col: 30, Type: 2, Value: ,
Row: 2, Col: 31, Type: 0, Value: CHAR
Row: 2, Col: 36, Type: 2, Value: *
Row: 2, Col: 38, Type: 0, Value: destFile
Row: 2, Col: 46, Type: 2, Value: )
Row: 2, Col: 47, Type: 2, Value: {
Row: 3, Col: 1, Type: 0, Value: FILE
Row: 4, Col: 8, Type: 2, Value: *
Row: 4, Col: 10, Type: 0, Value: file1
Row: 4, Col: 15, Type: 2, Value: =
Row: 4, Col: 17, Type: 0, Value: fopen
Row: 4, Col: 23, Type: 2, Value: (
Row: 4, Col: 24, Type: 0, Value: srcFile
Row: 4, Col: 31, Type: 2, Value: ,
Row: 4, Col: 32, Type: 4, Value: "r"
Row: 4, Col: 36, Type: 2, Value: )
Row: 4, Col: 37, Type: 2, Value: ;
Row: 4, Col: 38, Type: 0, Value: FILE
Row: 5, Col: 8, Type: 2, Value: *
Row: 5, Col: 10, Type: 0, Value: file2
Row: 5, Col: 15, Type: 2, Value: =
Row: 5, Col: 17, Type: 0, Value: fopen
Row: 5, Col: 23, Type: 2, Value: (
Row: 5, Col: 24, Type: 0, Value: destFile
```