Week3

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

Lab Exercise:

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

   and

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 for each identified token. The getNextToken() function must ignore tokens located within single-or multi-line comments, as well as those found inside string literals. Additionally, it should strip preprocessor directives.

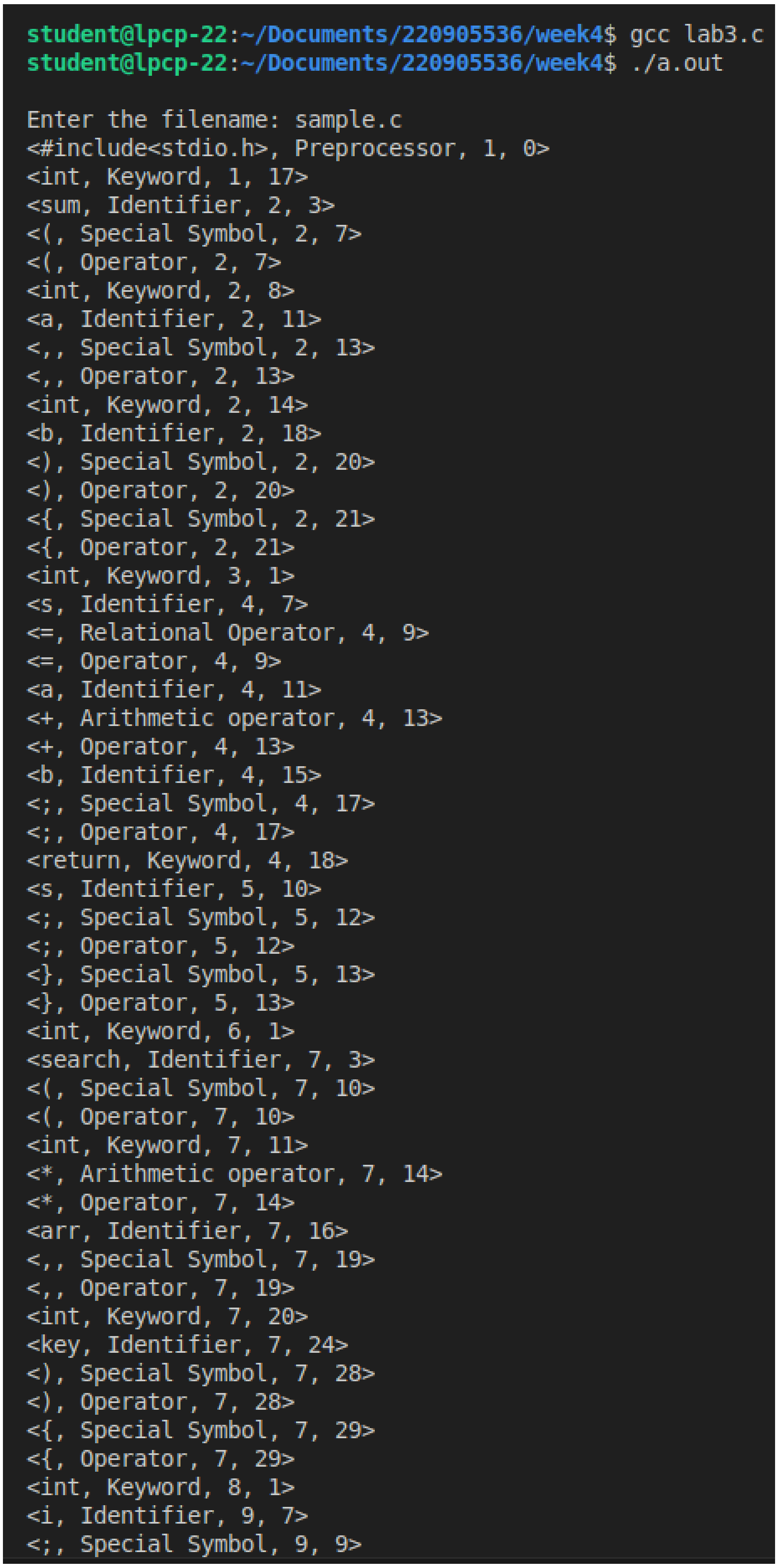
Source code of above 2 programs:

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

#define max_token_LENGTH 100

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;
}

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

const char* tokenTypeToString(TokenType type)
{
    switch (type)
    {
        case token_ID: return "Identifier";
        case token_NUMBER: return "Number";
        case token_OPERATOR: return "Operator";
        case token_KEYWORD: return "Keyword";
        case token_STRING: return "String";
        case token_COMMENT: return "Comment";
        case token_PREPROCESSOR: return "Preprocessor";
        case token_UNKNOWN: return "Unknown";
        case token_EOF: return "EOF";
        default: return "Invalid";
    }
}
```

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

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';

        char output[512];
        // Check for 2-character operators
        if (c == '+' || c == '-' || c == '!' || c == '<' || c == '>')
```

```
        {
            char next = fgetc(source);
            if (next == c || next == '=' || next == '-')
            { // Handle ++, --, ==, !=, <=, >=
                token.value[1] = next;
                token.value[2] = '\0';
            }
            else
            {
                ungetc(next, source); // Put back the character if it's not part of an operator
            }
        }

        // Now, print the classification
        if (isArithmeticOp(token.value))
        {
            snprintf(output, sizeof(output), "<%s, Arithmetic operator, %d, %d>\n", token.value,
token.row, token.col);
            printf("%s", output);
        }
        else if (isRelationOp(token.value))
        {
            snprintf(output, sizeof(output), "<%s, Relational Operator, %d, %d>\n", token.value,
token.row, token.col);
            printf("%s", output);
        }
        else if (isLogicalOp(token.value))
        {
            snprintf(output, sizeof(output), "<%s, Logical Operator, %d, %d>\n", token.value, token.row,
token.col);
            printf("%s", output);
        }
        else if (isNumericalConstant(token.value))
        {
            snprintf(output, sizeof(output), "<%s, num, %d, %d>\n", token.value, token.row, token.col);
            printf("%s", output);
        }
        else if (isSpecialSymbol(token.value))
        {
            snprintf(output, sizeof(output), "<%s, Special Symbol, %d, %d>\n", token.value, token.row,
token.col);
            printf("%s", output);
        }
        else if (isKeyword(token.value))
        {
            snprintf(output, sizeof(output), "<%s, Keyword, %d, %d>\n", token.value, token.row,
token.col);
            printf("%s", output);
        }
        else
        {
            snprintf(output, sizeof(output), "<%s, Identifier, %d, %d>\n", token.value, token.row,
token.col);
            printf("%s", output);
        }

        return token;
    }

    token.type = token_EOF;
    return token;
}

int main()
{
    char filename[128];
    printf("\nEnter the filename: ");
    scanf("%s", filename);
    int row = 1, col = 0;

    FILE *source = fopen(filename,"r");
    Token token = getNextToken(source, &row, &col);
    while (token.type != token_EOF)
    {
        printf("<%s, %s, %d, %d>\n", token.value, tokenTypeToString(token.type), token.row, token.col);
        token = getNextToken(source, &row, &col);
    }
    fclose(source);
    return 0;
}
```

Output:



```
student@lpcp-22:~/Documents/220905536/week4$ gcc lab3.c
student@lpcp-22:~/Documents/220905536/week4$ ./a.out

Enter the filename: sample.c
<#include<stdio.h>, Preprocessor, 1, 0>
<int, Keyword, 1, 17>
<sum, Identifier, 2, 3>
<(, Special Symbol, 2, 7>
<(, Operator, 2, 7>
<int, Keyword, 2, 8>
<a, Identifier, 2, 11>
<,, Special Symbol, 2, 13>
<,, Operator, 2, 13>
<int, Keyword, 2, 14>
<b, Identifier, 2, 18>
<), Special Symbol, 2, 20>
<), Operator, 2, 20>
<{, Special Symbol, 2, 21>
<{, Operator, 2, 21>
<int, Keyword, 3, 1>
<s, Identifier, 4, 7>
<=, Relational Operator, 4, 9>
<=, Operator, 4, 9>
<a, Identifier, 4, 11>
<+, Arithmetic operator, 4, 13>
<+, Operator, 4, 13>
<b, Identifier, 4, 15>
<;, Special Symbol, 4, 17>
<;, Operator, 4, 17>
<return, Keyword, 4, 18>
<s, Identifier, 5, 10>
<;, Special Symbol, 5, 12>
<;, Operator, 5, 12>
<}, Special Symbol, 5, 13>
<}, Operator, 5, 13>
<int, Keyword, 6, 1>
<search, Identifier, 7, 3>
<(, Special Symbol, 7, 10>
<(, Operator, 7, 10>
<int, Keyword, 7, 11>
<*, Arithmetic operator, 7, 14>
<*, Operator, 7, 14>
<arr, Identifier, 7, 16>
<,, Special Symbol, 7, 19>
<,, Operator, 7, 19>
<int, Keyword, 7, 20>
<key, Identifier, 7, 24>
<), Special Symbol, 7, 28>
<), Operator, 7, 28>
<{, Special Symbol, 7, 29>
<{, Operator, 7, 29>
<int, Keyword, 8, 1>
<i, Identifier, 9, 7>
<;, Special Symbol, 9, 9>
```