COMPILER DESIGN

ADITYA AGARWAL **ROLL NO. - 14** **REG NO. - 220905106**

LAB3 - CONSTRUCTION OF TOKEN GENERATOR

Q1 - Write functions to identify the following tokens.
a. Arithmetic, relational and logical operators.
b. Special symbols, keywords, numerical constants, string literals and identifiers.

CODE:

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

// Function to identify arithmetic operators
void identifyArithmeticOperators(char c) {
    if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%') {
        printf("Arithmetic Operator: %c\n", c);
    }
}

// Function to identify relational operators
void identifyRelationalOperators(char c, FILE *fp) {
    if (c == '=' || c == '<' || c == '>' || c == '!') {
        char nextChar = fgetc(fp);
        if (nextChar == '=') {
            printf("Relational Operator: %c%c\n", c, nextChar); // For operators like <=, >=, !=, ==
        } else {
            ungetc(nextChar, fp); // Push back the character if it's not part of a two-char operator
            printf("Relational Operator: %c\n", c); // For operators like <, >, =
        }
    }
}

// Function to identify logical operators
void identifyLogicalOperators(char c, FILE *fp) {
    if (c == '&' || c == '|') {
        char nextChar = fgetc(fp);
        if (nextChar == c) {
            printf("Logical Operator: %c%c\n", c, nextChar); // For && or ||
        } else {
            ungetc(nextChar, fp); // Push back the character if it's not part of a two-char operator
        }
    } else if (c == '!') {
        printf("Logical Operator: %c\n", c); // For !
    }
```

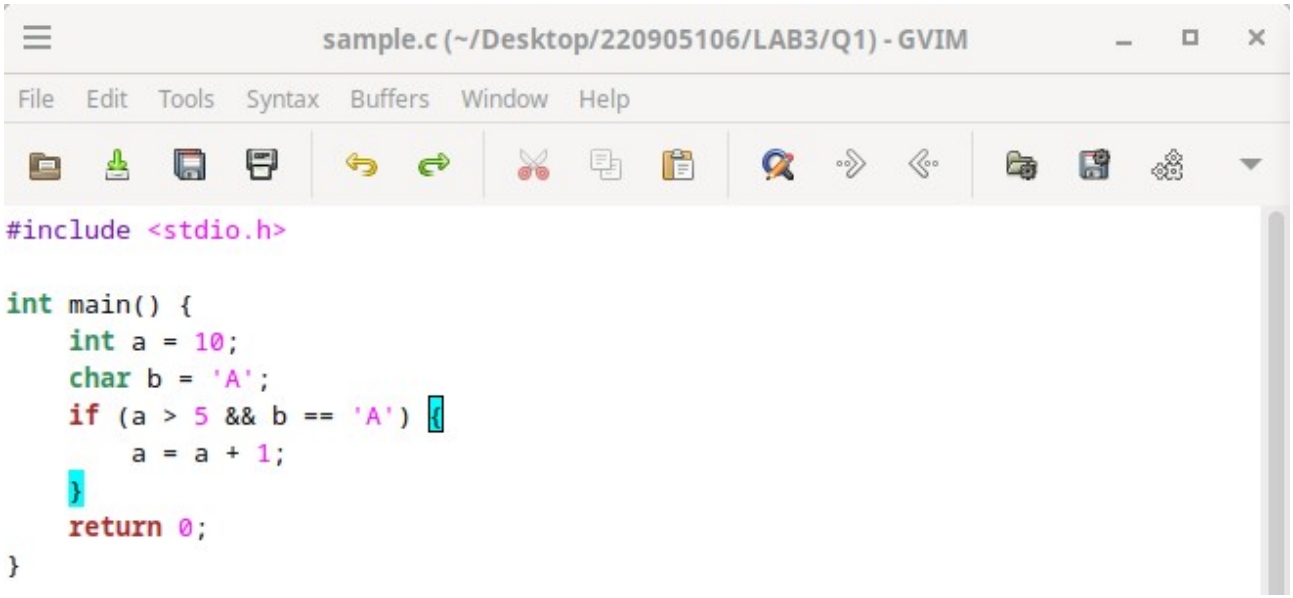
```
}

// Function to identify special symbols
void identifySpecialSymbols(char c) {
    if (c == '(') printf("Special Symbol: ( \n");
    if (c == ')') printf("Special Symbol: ) \n");
    if (c == '{') printf("Special Symbol: { \n");
    if (c == '}') printf("Special Symbol: } \n");
    if (c == '[') printf("Special Symbol: [ \n");
    if (c == ']') printf("Special Symbol: ] \n");
    if (c == ';') printf("Special Symbol: ; \n");
    if (c == ',') printf("Special Symbol: , \n");
    if (c == '#') printf("Special Symbol: # \n"); // Handle the '#' symbol as part of preprocessor
directives
}

// Function to identify preprocessor directives (like #include)
void identifyPreprocessorDirective(char *lexeme) {
    if (lexeme[0] == '#') {
        printf("Preprocessor Directive: %s\n", lexeme);
    }
}

// Function to identify keywords
void identifyKeywords(char *lexeme) {
    char *keywords[] = {"int", "char", "if", "else", "return", "for", "while", "void"};
    for (int i = 0; i < 8; i++) {
        if (strcmp(lexeme, keywords[i]) == 0) {
            printf("Keyword: %s\n", lexeme);
            return;
        }
    }
}

// Function to identify numerical constants
void identifyNumericalConstants(char *lexeme) {
    int isNumber = 1;
    for (int i = 0; lexeme[i] != '\0'; i++) {
        if (!isdigit(lexeme[i])) {
            isNumber = 0;
            break;
        }
    }
    if (isNumber) {
        printf("Numerical Constant: %s\n", lexeme);
    }
}

// Function to identify string literals
void identifyStringLiterals(char *lexeme) {
    if (lexeme[0] == '"' && lexeme[strlen(lexeme)-1] == '"') {
        printf("String Literal: %s\n", lexeme);
```

```
    }
}

// Function to identify character constants (e.g., 'A')
void identifyCharacterConstants(char *lexeme) {
    if (lexeme[0] == '\'' && lexeme[strlen(lexeme)-1] == '\'') {
        printf("Character Constant: %s\n", lexeme);
    }
}

// Function to identify identifiers
void identifyIdentifiers(char *lexeme) {
    if (isalpha(lexeme[0]) || lexeme[0] == '_') {
        int isValidIdentifier = 1;
        for (int i = 1; lexeme[i] != '\0'; i++) {
            if (!isalnum(lexeme[i]) && lexeme[i] != '_') {
                isValidIdentifier = 0;
                break;
            }
        }
        if (isValidIdentifier) {
            printf("Identifier: %s\n", lexeme);
        }
    }
}

// Main function to process the input file and identify all tokens
int main() {
    FILE *fp = fopen("sample.c", "r"); // Open the source file
    if (fp == NULL) {
        printf("Cannot open file\n");
        return 1;
    }

    char c;
    char lexeme[100];
    int lexemeIndex = 0;

    // Process each character of the input file
    while ((c = fgetc(fp)) != EOF) {
        if (isspace(c)) {
            continue;  // Skip whitespaces
        }

        // Handle preprocessor directive (#include, etc.)
        if (c == '#') {
            lexeme[lexemeIndex++] = c;
            while ((c = fgetc(fp)) != '\n' && c != EOF) {
                lexeme[lexemeIndex++] = c;
            }
            lexeme[lexemeIndex] = '\0';
            identifyPreprocessorDirective(lexeme);
```

```
    lexemeIndex = 0;
}

// Handle identifiers and keywords
else if (isalpha(c) || c == '_') {
    lexeme[lexemeIndex++] = c;
    while (isalnum(c = fgetc(fp)) || c == '_') {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex] = '\0';
    ungetc(c, fp);  // Push back the character after the identifier
    identifyKeywords(lexeme);
    identifyIdentifiers(lexeme);
    lexemeIndex = 0;
}

// Handle numerical constants
else if (isdigit(c)) {
    lexeme[lexemeIndex++] = c;
    while (isdigit(c = fgetc(fp))) {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex] = '\0';
    ungetc(c, fp);
    identifyNumericalConstants(lexeme);
    lexemeIndex = 0;
}

// Handle string literals
else if (c == '"') {
    lexeme[lexemeIndex++] = c;
    while ((c = fgetc(fp)) != '"' && c != EOF) {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex++] = c;
    lexeme[lexemeIndex] = '\0';
    identifyStringLiterals(lexeme);
    lexemeIndex = 0;
}

// Handle character literals (e.g., 'A')
else if (c == '\'') {
    lexeme[lexemeIndex++] = c;
    while ((c = fgetc(fp)) != '\'' && c != EOF) {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex++] = c;
    lexeme[lexemeIndex] = '\0';
    identifyCharacterConstants(lexeme);
    lexemeIndex = 0;
}
```

```
        // Handle special symbols and operators
        else {
            identifySpecialSymbols(c);
            identifyArithmeticOperators(c);
            identifyRelationalOperators(c, fp);
            identifyLogicalOperators(c, fp);
        }
    }

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

INPUT FILE :



OUTPUT:

CD_LAB_B1@debianpc-02: ~/Desktop/220905106/LAB3/Q1

File Edit View Search Terminal Help

```
Preprocessor Directive: #include <stdio.h>
Keyword: int
Identifier: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: int
Identifier: a
Relational Operator: =
Numerical Constant: 10
Special Symbol: ;
Keyword: char
Identifier: char
Identifier: b
Relational Operator: =
Character Constant: 'A'
Special Symbol: ;
Keyword: if
Identifier: if
Special Symbol: (
Identifier: a
Relational Operator: >
Numerical Constant: 5
Logical Operator: &&
Identifier: b
Relational Operator: ==
Character Constant: 'A'
Special Symbol: )
Special Symbol: {
Identifier: a
Relational Operator: =
Identifier: a
Arithmetic Operator: +
Numerical Constant: 1
Special Symbol: ;
Special Symbol: }
```

```
Special Symbol: ;
Special Symbol: }
Keyword: return
Identifier: return
Numerical Constant: 0
Special Symbol: ;
Special Symbol: }
CD_LAB_B1@debianpc-02:~/Desktop/220905106/LAB3/Q1$
```

Q2 - 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.

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

// Define token types for better readability
#define KEYWORD 1
#define IDENTIFIER 2
#define OPERATOR 3
#define NUMERIC_CONSTANT 4
#define STRING_LITERAL 5
#define PREPROCESSOR 6
#define COMMENT 7

// Token structure to store row, column, type, and value
typedef struct {
    int row;
    int col;
    int type;
    char value[100];
} Token;

// Function to check if the token is a keyword
int isKeyword(char *lexeme) {
    const char *keywords[] = {"int", "char", "if", "else", "return", "for", "while", "void"};
    for (int i = 0; i < 8; i++) {
        if (strcmp(lexeme, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Function to print token information
void printToken(Token t) {
    const char *types[] = {"Unknown", "Keyword", "Identifier", "Operator", "Numerical Constant",
"String Literal", "Preprocessor Directive", "Comment"};
    printf("Row: %d, Col: %d, Type: %s, Value: %s\n", t.row, t.col, types[t.type], t.value);
}

// Function to get the next token
Token getNextToken(FILE *fp, int *row, int *col) {
    Token t = {-1, -1, 0, ""}; // Initialize token
    char c;
    char lexeme[100];
```

```
int lexemeIndex = 0;
int inStringLiteral = 0;
int inComment = 0;

while ((c = fgetc(fp)) != EOF) {
    (*col)++;

    // Skip whitespace and handle newlines for row counting
    if (isspace(c)) {
        if (c == '\n') {
            (*row)++;
            *col = 0;
        }
        continue;
    }

    // Skip preprocessor directives
    if (c == '#' && *col == 1) {
        t.type = PREPROCESSOR;
        t.value[0] = c;
        while ((c = fgetc(fp)) != '\n' && c != EOF) {
            strncat(t.value, &c, 1);
        }
        (*row)++;
        (*col) = 0;
        return t;
    }

    // Handle comments
    if (!inStringLiteral && c == '/') {
        c = fgetc(fp);
        if (c == '/') {
            inComment = 1; // Single-line comment
            t.type = COMMENT;
            t.value[0] = '/';
            t.value[1] = '/';
            while ((c = fgetc(fp)) != '\n' && c != EOF) {
                strncat(t.value, &c, 1);
            }
            (*row)++;
            (*col) = 0;
            return t;
        } else if (c == '*') {
            inComment = 1; // Multi-line comment
            t.type = COMMENT;
            t.value[0] = '/';
            t.value[1] = '*';
            while ((c = fgetc(fp)) != EOF) {
                strncat(t.value, &c, 1);
                if (c == '*' && (c = fgetc(fp)) == '/') {
                    strncat(t.value, &c, 1);
                    break;
```

```
            }
        }
        (*row)++;
        (*col) = 0;
        return t;
    }
}

// Handle string literals
if (c == '"' && !inComment) {
    inStringLiteral = !inStringLiteral;
    t.type = STRING_LITERAL;
    t.value[0] = c;
    while ((c = fgetc(fp)) != '"' && c != EOF) {
        strncat(t.value, &c, 1);
    }
    t.value[strlen(t.value)] = '"'; // Closing quote
    (*row)++;
    (*col) = 0;
    return t;
}

// Handle keywords and identifiers
if (isalpha(c) || c == '_') {
    lexeme[lexemeIndex++] = c;
    while (isalnum(c = fgetc(fp)) || c == '_') {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex] = '\0';
    ungetc(c, fp); // Put the non-identifier character back
    t.row = *row;
    t.col = *col;
    if (isKeyword(lexeme)) {
        t.type = KEYWORD;
    } else {
        t.type = IDENTIFIER;
    }
    strcpy(t.value, lexeme);
    (*col) += lexemeIndex;
    return t;
}

// Handle numerical constants
if (isdigit(c)) {
    lexeme[lexemeIndex++] = c;
    while (isdigit(c = fgetc(fp))) {
        lexeme[lexemeIndex++] = c;
    }
    lexeme[lexemeIndex] = '\0';
    ungetc(c, fp);
    t.row = *row;
    t.col = *col;
```

```
            t.type = NUMERIC_CONSTANT;
            strcpy(t.value, lexeme);
            (*col) += lexemeIndex;
            return t;
        }

        // Handle operators and special symbols
        if (strchr("+-*/=<>!&|^%,;(){}", c)) {
            t.row = *row;
            t.col = *col;
            t.type = OPERATOR;
            t.value[0] = c;
            t.value[1] = '\0';
            return t;
        }
    }

    t.row = *row;
    t.col = *col;
    t.type = 0;
    strcpy(t.value, "EOF");
    return t;
}

int main() {
    FILE *fp = fopen("sample.c", "r"); // Open the source file
    if (fp == NULL) {
        printf("Cannot open file\n");
        return 1;
    }

    int row = 1, col = 1;
    Token t;

    while ((t = getNextToken(fp, &row, &col)).type != 0) {
        printToken(t); // Print the token details
    }

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

INPUT FILE :

sample.c (~/Desktop/220905106/LAB3/Q2) - GVIM1

File Edit Tools Syntax Buffers Window Help

```c
#include <stdio.h>

int main() {
    int a = 10;
    char b = 'A';
    if (a > 5 && b == 'A') {
        a = a + 1;
    }
    return 0;
}
```

OUTPUT:

```
CD_LAB_B1@debianpc-02:~/Desktop$ ./q2
Row: 1, Col: 3, Type: Identifier, Value: include
Row: 1, Col: 12, Type: Operator, Value: <
Row: 1, Col: 13, Type: Identifier, Value: stdio
Row: 1, Col: 20, Type: Identifier, Value: h
Row: 1, Col: 22, Type: Operator, Value: >
Row: 3, Col: 1, Type: Keyword, Value: int
Row: 3, Col: 6, Type: Identifier, Value: main
Row: 3, Col: 11, Type: Operator, Value: (
Row: 3, Col: 12, Type: Operator, Value: )
Row: 3, Col: 14, Type: Operator, Value: {
Row: 4, Col: 5, Type: Keyword, Value: int
Row: 4, Col: 10, Type: Identifier, Value: a
Row: 4, Col: 13, Type: Operator, Value: =
Row: 4, Col: 15, Type: Numerical Constant, Value: 10
Row: 4, Col: 18, Type: Operator, Value: ;
Row: 5, Col: 5, Type: Keyword, Value: char
Row: 5, Col: 11, Type: Identifier, Value: b
Row: 5, Col: 14, Type: Operator, Value: =
Row: 5, Col: 17, Type: Identifier, Value: A
Row: 5, Col: 20, Type: Operator, Value: ;
Row: 6, Col: 5, Type: Keyword, Value: if
Row: 6, Col: 9, Type: Operator, Value: (
Row: 6, Col: 10, Type: Identifier, Value: a
Row: 6, Col: 13, Type: Operator, Value: >
Row: 6, Col: 15, Type: Numerical Constant, Value: 5
Row: 6, Col: 18, Type: Operator, Value: &
Row: 6, Col: 19, Type: Operator, Value: &
Row: 6, Col: 21, Type: Identifier, Value: b
Row: 6, Col: 24, Type: Operator, Value: =
Row: 6, Col: 25, Type: Operator, Value: =
Row: 6, Col: 28, Type: Identifier, Value: A
Row: 6, Col: 31, Type: Operator, Value: )
Row: 6, Col: 33, Type: Operator, Value: {
Row: 7, Col: 9, Type: Identifier, Value: a
Row: 7, Col: 12, Type: Operator, Value: =
Row: 7, Col: 14, Type: Identifier, Value: a
Row: 7, Col: 17, Type: Operator, Value: +
Row: 7, Col: 17, Type: Operator, Value: +
Row: 7, Col: 19, Type: Numerical Constant, Value: 1
Row: 7, Col: 21, Type: Operator, Value: ;
Row: 8, Col: 5, Type: Operator, Value: }
Row: 9, Col: 5, Type: Keyword, Value: return
Row: 9, Col: 13, Type: Numerical Constant, Value: 0
Row: 9, Col: 15, Type: Operator, Value: ;
Row: 10, Col: 1, Type: Operator, Value: }
CD_LAB_B1@debianpc-02:~/Desktop$
```