COMPILER DESIGN LAB 3

Name: Devadathan N R

Registration No: 230905010

Class: CSE - A1

Roll No: 04

Date: 20/01/2026

Question 1 & 2:

Design a lexical analyzer with a getNextToken() function to process 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 within single-line or multi-line comments and those inside string literals. It should also strip out preprocessor directives.

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

#define BUFFER_SIZE 4096
#define MAX_LEXEME 100
```

```
struct token{
    char lexeme[MAX_LEXEME];
    char type[20];
    int row;
    int col;
};

static char buffer[BUFFER_SIZE];
static int bufSize=0;
static int LB=0,FP=0;
static int row=1,col=1;
static int initialized=0;

const char *keywords[]={
 "int","float","double","char","void","long","short","signed","unsigned",
 "if","else","while","for","do","switch","case","default",
 "break","continue","return","goto",
 "auto","register","static","extern","const","volatile",
 "struct","union","enum","typedef","sizeof",
 "printf","scanf","malloc","free","strlen","strcpy","strcmp","exit",
 "main"
};

int keywordCount = sizeof(keywords)/sizeof(keywords[0]);

void loadBuffer(FILE *fp){
    bufSize=fread(buffer,1,BUFFER_SIZE-1,fp);
    buffer[bufSize]='\0';
}

char advanceChar(){
    char c=buffer[FP++];
    if(c=='\n'){ row++; col=1; }
    else col++;
    return c;
}
```

```
char peekChar(){ return buffer[FP]; }

int isKeyword(char *str){
   for(int i=0;i<keywordCount;i++)
      if(strcmp(str,keywords[i])==0) return 1;
   return 0;
}

int isSpecialSymbol(char c){
   return (c=='('||c==')'||c=='{'||c=='}'|| c=='['||c==']'||c==';'||c==',');
}

void skipWhiteSpace(){
   while(peekChar()==' '||peekChar()=='\t'||peekChar()=='\n') advanceChar();
}

void skipSingleLineComment(){
   while(advanceChar()!='\n');
}

void skipMultiLineComment(){
   char c;
   while(FP<bufSize){
      c=advanceChar();
      if(c=='*' && peekChar()=='/'){
         advanceChar();
         break;
      }
   }
}

void skipPreprocessor(){
   while(advanceChar()!='\n');
}
```

```
struct token getNextToken(FILE *fp){
    struct token tk;
    if(!initialized){
        loadBuffer(fp);
        initialized=1;
    }

start:
    skipWhiteSpace();

    tk.row=row;
    tk.col=col;

    LB=FP;
    char c=advanceChar();

    if(c=='\0'){
        strcpy(tk.type,"EOF");
        return tk;
    }

    if(c=='#'){
        skipPreprocessor();
        goto start;
    }

    if(c=='/'){
        if(peekChar()=='/'){
            advanceChar();
            skipSingleLineComment();
            goto start;
        }
        if(peekChar()=='*'){
            advanceChar();
            skipMultiLineComment();
            goto start;
```

```
    }
}

if(c=='"'){
    while(peekChar()!='"' && peekChar()!='\0')advanceChar();
    advanceChar();

    int len=FP-LB;
    strncpy(tk.lexeme,buffer+LB,len);
    tk.lexeme[len]='\0';
    strcpy(tk.type,"STRING_LITERAL");
    return tk;
}

if(isalpha(c)||c=='_'){
    while(isalnum(peekChar())||peekChar()=='_')
        advanceChar();

    int len=FP-LB;
    strncpy(tk.lexeme,buffer+LB,len);
    tk.lexeme[len]='\0';

    if(isKeyword(tk.lexeme)) strcpy(tk.type,"KEYWORD");
    else strcpy(tk.type,"ID");
    return tk;
}

if(isdigit(c)){
    while(isdigit(peekChar())) advanceChar();
    if(peekChar()=='.'){
        advanceChar();
        while(isdigit(peekChar())) advanceChar();
    }

    int len=FP-LB;
    strncpy(tk.lexeme,buffer+LB,len);
```

```
    tk.lexeme[len]='\0';
    strcpy(tk.type,"NUM");
    return tk;
}

if(c=='=' && peekChar()=='='){
    advanceChar();
    strcpy(tk.lexeme,"==");
    strcpy(tk.type,"REL_OP");
    return tk;
}

if(c=='!' && peekChar()=='='){
    advanceChar();
    strcpy(tk.lexeme,"!=");
    strcpy(tk.type,"REL_OP");
    return tk;
}

if(c=='<' && peekChar()=='='){
    advanceChar();
    strcpy(tk.lexeme,"<=");
    strcpy(tk.type,"REL_OP");
    return tk;
}

if(c=='>' && peekChar()=='='){
    advanceChar();
    strcpy(tk.lexeme,">=");
    strcpy(tk.type,"REL_OP");
    return tk;
}

if((c=='&'&&peekChar()=='&')||(c=='|'&&peekChar()=='|')){
    advanceChar();
    int len=FP-LB;
```

```
    strncpy(tk.lexeme,buffer+LB,len);
    tk.lexeme[len]='\0';
    strcpy(tk.type,"LOGICAL_OP");
    return tk;
}

if(c=='+'||c=='-'||c=='*'||c=='/'||c=='%'){
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"ARITH_OP");
    return tk;
}

if(c=='<'||c=='>'){
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"REL_OP");
    return tk;
}

if(c=='!'){
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"LOGICAL_OP");
    return tk;
}

if(c=='='){
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"ASSIGN_OP");
    return tk;
}

if(isSpecialSymbol(c)){
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"SYMBOL");
    return tk;
}
```

```
    tk.lexeme[0]=c; tk.lexeme[1]='\0';
    strcpy(tk.type,"UNKNOWN");
    return tk;
}

int main(){
    FILE *fp=fopen("sample.c","r");
    if(!fp){
        printf("File cannot be opened\n");
        return 0;
    }

    struct token tk;
    printf("\n----- TOKENS-----\n");

    while(1){
        tk=getNextToken(fp);
        if(strcmp(tk.type,"EOF")==0) break;

        printf("<%s, %s, Row: %d, Col: %d>\n",
              tk.lexeme,tk.type,tk.row,tk.col);
    }

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

Input

sample.c

```
#include <stdio.h>
#define MAX 100

// single line comment
```

```
/*
   multi
   line
   comment
*/

int main() {

    float a=10,b=20;
    double result = a+b;

    if (a <= b && b != 0 || a == 10) {
        if (a >= b && a != b) {
            return 1;
        }
    }

    printf("A string literal");

    return 0;
}
```

Token Types;

```
KEYWORD

ID

NUM

STRING_LITERAL

ARITH_OP
```

```
REL_OP

LOGICAL_OP

ASSIGN_OP

SYMBOL
```

Output

```
CD_A1@CL3-02:~/Desktop/230905010_DevadathanNR_CS_A1/03_LAB$
CD_A1@CL3-02:~/Desktop/230905010_DevadathanNR_CS_A1/03_LAB$

----- TOKENS-----
<int, KEYWORD, Row: 12, Col: 1>
<main, KEYWORD, Row: 12, Col: 5>
<(, SYMBOL, Row: 12, Col: 9>
<), SYMBOL, Row: 12, Col: 10>
<{, SYMBOL, Row: 12, Col: 12>
<float, KEYWORD, Row: 14, Col: 5>
<a, ID, Row: 14, Col: 11>
<=, ASSIGN_OP, Row: 14, Col: 12>
<10, NUM, Row: 14, Col: 13>
<,, SYMBOL, Row: 14, Col: 15>
<b, ID, Row: 14, Col: 16>
<=, ASSIGN_OP, Row: 14, Col: 17>
<20, NUM, Row: 14, Col: 18>
<;, SYMBOL, Row: 14, Col: 20>
<double, KEYWORD, Row: 15, Col: 5>
<result, ID, Row: 15, Col: 12>
<=, ASSIGN_OP, Row: 15, Col: 19>
<a, ID, Row: 15, Col: 21>
<+, ARITH_OP, Row: 15, Col: 22>
<b, ID, Row: 15, Col: 23>
<;, SYMBOL, Row: 15, Col: 24>
<if, KEYWORD, Row: 17, Col: 5>
<(, SYMBOL, Row: 17, Col: 8>
<a, ID, Row: 17, Col: 9>
<<=, REL_OP, Row: 17, Col: 11>
<b, ID, Row: 17, Col: 14>
<&&, LOGICAL_OP, Row: 17, Col: 16>
<b, ID, Row: 17, Col: 19>
<!=, REL_OP, Row: 17, Col: 21>
<0, NUM, Row: 17, Col: 24>
<||, LOGICAL_OP, Row: 17, Col: 26>
<a, ID, Row: 17, Col: 29>
<==, REL_OP, Row: 17, Col: 31>
<10, NUM, Row: 17, Col: 34>
<), SYMBOL, Row: 17, Col: 36>
<{, SYMBOL, Row: 17, Col: 38>
<if, KEYWORD, Row: 18, Col: 6>
<(, SYMBOL, Row: 18, Col: 9>
<a, ID, Row: 18, Col: 10>
<>=, REL_OP, Row: 18, Col: 12>
<b, ID, Row: 18, Col: 15>
<&&, LOGICAL_OP, Row: 18, Col: 17>
<a, ID, Row: 18, Col: 20>
<!=, REL_OP, Row: 18, Col: 22>
<b, ID, Row: 18, Col: 25>
<), SYMBOL, Row: 18, Col: 26>
<{, SYMBOL, Row: 18, Col: 28>
<return, KEYWORD, Row: 19, Col: 10>
<1, NUM, Row: 19, Col: 17>
<;, SYMBOL, Row: 19, Col: 18>
<}, SYMBOL, Row: 20, Col: 6>
<}, SYMBOL, Row: 21, Col: 5>
<printf, KEYWORD, Row: 23, Col: 5>
<(, SYMBOL, Row: 23, Col: 11>
<"A string literal", STRING_LITERAL, Row: 23, Col: 12>
<), SYMBOL, Row: 23, Col: 30>
<;, SYMBOL, Row: 23, Col: 31>
<return, KEYWORD, Row: 25, Col: 5>
<0, NUM, Row: 25, Col: 12>
<;, SYMBOL, Row: 25, Col: 13>
<}, SYMBOL, Row: 26, Col: 1>
```

[Text Format]

```
----- TOKENS-----
<int, KEYWORD, Row: 12, Col: 1>
<main, KEYWORD, Row: 12, Col: 5>
<(, SYMBOL, Row: 12, Col: 9>
<), SYMBOL, Row: 12, Col: 10>
<{, SYMBOL, Row: 12, Col: 12>
<float, KEYWORD, Row: 14, Col: 5>
<a, ID, Row: 14, Col: 11>
<=, ASSIGN_OP, Row: 14, Col: 12>
<10, NUM, Row: 14, Col: 13>
<,, SYMBOL, Row: 14, Col: 15>
<b, ID, Row: 14, Col: 16>
<=, ASSIGN_OP, Row: 14, Col: 17>
<20, NUM, Row: 14, Col: 18>
<;, SYMBOL, Row: 14, Col: 20>
<double, KEYWORD, Row: 15, Col: 5>
<result, ID, Row: 15, Col: 12>
<=, ASSIGN_OP, Row: 15, Col: 19>
<a, ID, Row: 15, Col: 21>
<+, ARITH_OP, Row: 15, Col: 22>
<b, ID, Row: 15, Col: 23>
<;, SYMBOL, Row: 15, Col: 24>
<if, KEYWORD, Row: 17, Col: 5>
<(, SYMBOL, Row: 17, Col: 8>
<a, ID, Row: 17, Col: 9>
<<=, REL_OP, Row: 17, Col: 11>
<b, ID, Row: 17, Col: 14>
<&&, LOGICAL_OP, Row: 17, Col: 16>
<b, ID, Row: 17, Col: 19>
<!=, REL_OP, Row: 17, Col: 21>
<0, NUM, Row: 17, Col: 24>
<||, LOGICAL_OP, Row: 17, Col: 26>
```

```
<a, ID, Row: 17, Col: 29>
⇐, REL_OP, Row: 17, Col: 31>
<10, NUM, Row: 17, Col: 34>
<), SYMBOL, Row: 17, Col: 36>
<{, SYMBOL, Row: 17, Col: 38>
<if, KEYWORD, Row: 18, Col: 6>
<(, SYMBOL, Row: 18, Col: 9>
<a, ID, Row: 18, Col: 10>
<>=, REL_OP, Row: 18, Col: 12>
<b, ID, Row: 18, Col: 15>
<&&, LOGICAL_OP, Row: 18, Col: 17>
<a, ID, Row: 18, Col: 20>
<!=, REL_OP, Row: 18, Col: 22>
<b, ID, Row: 18, Col: 25>
<), SYMBOL, Row: 18, Col: 26>
<{, SYMBOL, Row: 18, Col: 28>
<return, KEYWORD, Row: 19, Col: 10>
<1, NUM, Row: 19, Col: 17>
<;, SYMBOL, Row: 19, Col: 18>
<}, SYMBOL, Row: 20, Col: 6>
<}, SYMBOL, Row: 21, Col: 5>
<printf, KEYWORD, Row: 23, Col: 5>
<(, SYMBOL, Row: 23, Col: 11>
<"A string literal", STRING_LITERAL, Row: 23, Col: 12>
<), SYMBOL, Row: 23, Col: 30>
<;, SYMBOL, Row: 23, Col: 31>
<return, KEYWORD, Row: 25, Col: 5>
<0, NUM, Row: 25, Col: 12>
<;, SYMBOL, Row: 25, Col: 13>
<}, SYMBOL, Row: 26, Col: 1>
```