CD Lab #4

Question 1

```
/*
Design a lexical analyzer which contains getNextToken( ) for a simple C program to
create a structure of token each time and return, which includes row number, column
number and token type.

The tokens to be identified are arithmetic operators, relational
operators, logical operators, special symbols, keywords, numerical constants, string
literals and identifiers.

Also, getNextToken() should ignore all the tokens when
encountered inside single line or multiline comment or inside string literal. Preprocessor
directive should also be stripped.
*/

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

#define FILEINPUT "input.c"
#define TableLength 150
int count=0;

enum tokenType {
   EOFILE=-1,
   LESS_THAN, LESS_THAN_OR_EQUAL,
   GREATER_THAN, GREATER_THAN_OR_EQUAL,
   EQUAL,NOT_EQUAL
};

// creating basic token structure
struct token {
        char lexeme [64];
        int index,row,col; //Line numbers.
        char type[20];
   enum tokenType etype;
};

struct ST {
   char lexeme[64];
   char type[5];
   int size;
};

const char *identifiers[] = {"int", "char", "func"};

// // creating Linked List structure
// struct listToken {
//     struct token tkn;
```

```
//     struct listToken *next;
// };

// Initializing Table
struct ST Table [TableLength];


// // HELPER FUNCTIONS FOR TOKEN STRUCTURE
static int row=1,col=1;
char buf[2048];

const char specialsymbols[] = {'?',';',':',','};
const char *keywords[] = {"const", "char", "int","return","for", "while", "do",
                          "switch", "if", "else","unsigned", "case", "break"};

const char arithmeticsymbols[]= {'*', '+', '-', '/'};


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

int charBelongsTo(int c,const char *arr) {
   int len;

   if(arr==specialsymbols) {
      len=sizeof(specialsymbols)/sizeof(char);
   }

   else if(arr==arithmeticsymbols) {
      len=sizeof(arithmeticsymbols)/sizeof(char);
   }

   for(int i=0; i<len; i++) {
      if(c==arr[i]) {
         return 1;
      }
   }

   return 0;
}

void fillToken(struct token *tkn, char c, int index, int row, int col, char *type) {
   tkn->row=row;
   tkn->col=col;
   tkn->index = index;
```

```
   strcpy(tkn->type,type);
   tkn->lexeme[0]=c;
   tkn->lexeme[1]='\0';
}

void newLine() {
   ++row;
   col=1;
}

void remComments(char src_dir[], char res_dir[]){
        // declaring file pointers and variables
        FILE *fa, *fa_, *fb;
        int ca, ca_, temp;

        // updating res_dir
   strcpy(res_dir, src_dir);
        temp = strlen(res_dir);
        res_dir[temp-2] =  '\0';
        strcat(res_dir, "_remCom.c");

   // printf("source : %s, result : %s\n", src_dir, res_dir);

        // opening files
        fa = fopen(src_dir, "r");
        fa_ = fopen(src_dir, "r");
        fb = fopen(res_dir, "w+");

   // initializing ca
   ca = getc(fa);

   // initializing ca_
   ca_ = getc(fa_);
   ca_ = getc(fa_);

        // going through the document
        while(ca != EOF){
                if(ca == ' ' && ca_ == ' '){ // if more than 2 consecutive blanks are found
                        ca = getc(fa);
                        ca_ = getc(fa_);
                        continue;
                } else if (ca == '\t' && ca_ == '\t') { // if more than 2 consecutive tabs are found
                        ca = getc(fa);
                        ca_ = getc(fa_);
                        continue;
                } else if (ca == '\t' && ca_ != '\t') { // condensing consecutive tabs
                        putc(' ', fb);
                        ca = getc(fa);
                        ca_ = getc(fa_);
                } else { // writing all other characters into the file
                        putc(ca, fb);
                        ca = getc(fa);
```

```
                        ca_ = getc(fa_);
                }
        }

        // closing files
        fclose(fa);
        fclose(fa_);
        fclose(fb);

}

void remDirectives(char src_dir[], char res_dir[]){
        // declaring file pointers and variables
        FILE *fa, *fa_, *fb;
        int ca, ca_, temp, flag_1, flag_2;

        // updating res_dir
    strcpy(res_dir, src_dir);
    temp = strlen(res_dir);
    res_dir[temp-2] =  '\0';
    strcat(res_dir, "_remDirec.c");

    // printf("source : %s, result : %s\n", src_dir, res_dir);

        // opening files
        fa = fopen(src_dir, "r");
        fa_ = fopen(src_dir, "r");
        fb = fopen(res_dir, "w+");

    // printf("++\n");

        // initializing ca
        ca = getc(fa);

        // initializing ca_
        ca_ = getc(fa_);

        // going through the document
        while(ca != EOF){

                // check if '#' is encountered
                if(ca == '#'){

                        // check if it is actually a directive
                        while(ca_ != '\n'){
                                if(ca_ == '<') flag_1 = 1;
                                if(ca_ == '>') flag_2 = 1;
                                ca_ = getc(fa_);
                        }

                        ca_ = getc(fa_);
```

```
                        if(flag_1==1 && flag_2==1){                 // removing directive
                                flag_2 = 0;
                                flag_1 = 0;

                                while(ca != '\n'){
                                        ca = getc(fa);
                                }

                                ca = getc(fa);
                                continue;
                        } else {                                            // keeping sentence with #
                                flag_2 = 0;
                                flag_1 = 0;

                                while(ca != '\n'){
                                        putc(ca, fb);
                                        ca = getc(fa);
                                }

                                putc(ca, fb);
                                ca = getc(fa);
                                continue;
                        }
                }

                // keeping all other charaters
                putc(ca, fb);
                ca = getc(fa);
                ca_ = getc(fa_);
        }

        // closing files
        fclose(fa);
        fclose(fa_);
        fclose(fb);
}

void remLines(char src_dir[], char res_dir[]){
   // printf("==\n");
        remComments(src_dir, res_dir);

   // printf("==\n");
   strcpy(src_dir, res_dir);

   // printf("==\n");
        remDirectives(src_dir, res_dir);

   // printf("==\n");
}
```

```
struct token getNextToken(FILE *f1){
        // count line numer and column number [ USE WEEK1 Q1 ]

        // identify tokens nd get them into the token data structure [USE WEEK2 Q3]


        int c;
    struct token tkn= {
       .row=-1
    };

    int gotToken=0;

    while(!gotToken && (c=fgetc(f1))!=EOF) {

        // checking for Special Symbols
        if(charBelongsTo(c,specialsymbols)) {
        fillToken(&tkn, c, count, row, col, "SS");
        count = count + 1;
        gotToken=1;
        ++col;
      }

        // checking for Arithmatic Symbols
      else if(charBelongsTo(c,arithmeticsymbols)) {
        fillToken(&tkn,c,count,row,col,"ARITHMETIC OPERATOR");
        count++;
        gotToken=1;
        ++col;
      }

      // // HANDLING BRACKETS
      // left bracket
      else if(c=='(') {
        fillToken(&tkn,c,count,row,col,"LB");
        count++;
        gotToken=1;
        ++col;
      }

      // right bracket
      else if(c==')') {
        fillToken(&tkn,c,count,row,col,"RB");
        count++;
        gotToken=1;
        ++col;
      }

      // left curly bracket
      else if(c=='{') {
        fillToken(&tkn,c,count,row,col,"LC");
        count++;
```

```
    gotToken=1;
    ++col;
}

// right curly bracket
else if(c=='}') {
    fillToken(&tkn,c,count,row,col,"RC");
    count++;
    gotToken=1;
    ++col;
}

// // HANDLING DOUBLE OPERATORS
// checking for '++'
else if(c=='+') {
    int d=fgetc(f1);

    if(d!='+') {
        fillToken(&tkn,c,count,row,col,"ARITHMETIC OPERATOR");
        count++;
        gotToken=1;
        ++col;
        fseek(f1,-1,SEEK_CUR);
    }

    else {
        fillToken(&tkn,c,count,row,col,"UNARY OPERATOR");
        count++;
        strcpy(tkn.lexeme,"++");
        gotToken=1;
        col+=2;
    }
}

// checking for '--'
else if(c=='-') {
    int d=fgetc(f1);

    if(d!='-') {
        fillToken(&tkn,c,count,row,col,"ARITHMETIC OPERATOR");
        count++;
        gotToken=1;
        ++col;
        fseek(f1,-1,SEEK_CUR);
    }

    else {
        fillToken(&tkn,c,count,row,col,"UNARY OPERATOR");
        count++;
        strcpy(tkn.lexeme,"--");
        gotToken=1;
        col+=2;
```

```
    }
}

// checking for '=='
else if(c=='=') {
    int d=fgetc(f1);

    if(d!='=') {
        fillToken(&tkn,c,count,row,col,"ASSIGNMENT OPERATOR");
        count++;
        gotToken=1;
        ++col;
        fseek(f1,-1,SEEK_CUR);
    }

    else {
        fillToken(&tkn,c,count,row,col,"RELATIONAL OPERATOR");
        count++;
        strcpy(tkn.lexeme,"==");
        gotToken=1;
        col+=2;
    }
}


// handling numbers
else if(isdigit(c)) {
    tkn.row=row;
    tkn.col=col++;
    tkn.lexeme[0]=c;

    int k=1;

    while((c=fgetc(f1))!=EOF && isdigit(c)) {
        tkn.lexeme[k++]=c;
        col++;
    }

    tkn.lexeme[k]='\0';
    strcpy(tkn.type,"NUMBER");
    tkn.index = count++;
    gotToken=1;
    fseek(f1,-1,SEEK_CUR);
}

// handling newlines
else if(c=='\n') {
    newLine();
    c = fgetc(f1);

    if(c == '#') {
        while((c = fgetc(f1)) != EOF && c != '\n');
```

```
        newLine();
    }

    else if(c != EOF) {
        fseek(f1, -1, SEEK_CUR);
    }
}

// handling whitespace
else if(isspace(c)) {
    ++col;
}

// handling Alpha-Numeric Data
else if(isalpha(c)||c=='_') {
    tkn.row=row;
    tkn.col=col++;
    tkn.lexeme[0]=c;
    int k=1;

    while((c=fgetc(f1))!= EOF && isalnum(c)) {
        tkn.lexeme[k++]=c;
        ++col;
    }

    tkn.lexeme[k]='\0';

    if(isKeyword(tkn.lexeme)) {
        strcpy(tkn.type,"KEYWORD");
    } else if (c=fgetc(f1) == '(') {
        strcpy(tkn.type,"FUNCTION");
    } else {
        strcpy(tkn.type,"IDENTIFIER");
    }

    tkn.index = count++;

    gotToken=1;
    fseek(f1,-1,SEEK_CUR);
}

// handling comments
else if(c=='/') {
    int d=fgetc(f1);
    ++col;//Do we check EOF here?

    if(d=='/') {
        while((c=fgetc(f1))!= EOF && c!='\n') {
            ++col;
        }

        if(c=='\n') {
```

```
            newLine();
        }
    }

    else if(d=='*') {
        do {
            if(d=='\n') {
                newLine();
            }

            while((c==fgetc(f1))!= EOF && c!='*') {
                ++col;

                if(c=='\n') {
                    newLine();
                }
            }

            ++col;
        } while((d==fgetc(f1))!= EOF && d!='/' && (++col));
        ++col;
    }

    else {
        fillToken(&tkn,c,count,row,--col,"ARITHMETIC OPERATOR");
        count++;
        gotToken=1;
        fseek(f1,-1,SEEK_CUR);
    }
}

// handling string operators
else if(c == '"') {
    tkn.row = row;
    tkn.col = col;
    strcpy(tkn.type, "STRING LITERAL");
    tkn.index = count++;
    int k = 1;
    tkn.lexeme[0] = '"';

    while((c = fgetc(f1)) != EOF && c != '"') {
        tkn.lexeme[k++] = c;
        ++col;
    }

    tkn.lexeme[k] = '"';
    gotToken = 1;
}

// handling Relational Operators
else if(c == '<' || c == '>' || c == '!') {
    fillToken(&tkn, c, count, row, col, "RELATIONAL OPERATOR");
```

```
            count++;
            ++col;
            int d = fgetc(f1);
            if(d == '=') {
               ++col;
               strcat(tkn.lexeme, "=");
            }

            else {
               if(c == '!') {
                  strcpy(tkn.type, "LOGICAL OPERATOR");
               }

               fseek(f1, -1, SEEK_CUR);
            }

            gotToken = 1;
         }

         // handling 'and', 'or' operators
         else if(c == '&' || c == '|') {
            int d = fgetc(f1);

            if(c == d) {
               tkn.lexeme[0] = tkn.lexeme[1] = c;
               tkn.lexeme[2] = '\0';
               tkn.row = row;
               tkn.col = col;
               ++col;
               gotToken = 1;
               strcpy(tkn.type, "LOGICAL OPERATOR");
            }

            else {
               fseek(f1, -1, SEEK_CUR);
            }
            ++col;
         }

         // going to next column if unexpected literal gotten
         else {
            ++col;
         }
      }

      return tkn;
}

void printToken(struct token tkn){
            printf("<%d, %s, %d, %d>\n",tkn.index, tkn.type, tkn.row, tkn.col);
}
```

```
void printSymbol(struct ST tkn){
    printf("<%s, %s, %d>\n",tkn.lexeme, tkn.type, tkn.size);
}

// // HELPER FUNCTIONS FOR LINKED LIST
// initializing table as null
// void initialize(){
//     for(int i=0; i<TableLength; i++){
//         Table[i] = NULL;
//     }
// }

// function to display list
void Display(){
    int i=0;
    printf("=== Our Tabel is ===\n");

    if(Table[0].lexeme == NULL){
        printf("*** The Table is Empty ***\n\n\n");
        return;
    }

    while(Table[i].size != 0){
        // && Table[i].index <= TableLength
        printSymbol(Table[i]);
        i++;
    }
}

// function to develop OpenHash function on string
int HASH(){

}

// function to check if a lexeme exists in a token (return 1 if found)
int SEARCH(){

}

// inserting a new token to our Table
void INSERT(struct token tkn, int index){
    // struct ST symbol;

    if(!strcmp(tkn.type, "NUMBER")){
        strcpy(Table[index].lexeme, tkn.lexeme);
        Table[index].size = sizeof(int);
        strcpy(Table[index].type, "int");

    } else if (strcmp(tkn.type, "STRING LITERAL")) {
        strcpy(Table[index].lexeme, tkn.lexeme);
        Table[index].size = sizeof(Table[index].lexeme);
        strcpy(Table[index].type, "char");
```

```
    } else if (strcmp(tkn.type,"FUNCTION")) {
        strcpy(Table[index].lexeme, tkn.lexeme);
        Table[index].size = 0;
        strcpy(Table[index].type, "func");

    }

    // printf("<%s, %s, %d>\n",symbol.lexeme, symbol.type, symbol.size);
    // Table[index] = &symbol;

}

void lexicalAnalyzer(char dir[]){

            FILE *f1=fopen(dir,"r");
    if(f1==NULL) {
        printf("Error! File cannot be opened!\n");
        return;
    }

    // initializing our token Table
    // initialize();

    // printing empty Table
    // Display();

    // printf("//\n");

    int i=0;
            struct token tkn;
    while((tkn=getNextToken(f1)).row!=-1){
        INSERT(tkn, i);
        i++;
        // printToken(tkn);
    }

    // printf("//\n");

    // printing full Table
    Display();

    // printf("//\n");

    fclose(f1);
}


int main(){
            // printf("..\n");
            char inputFile[30] = "input.c", outputFile[30] = "";
```

```
    // printf("..\n");
    // removing comments and preprocessor drives [ USE WEEK2 Q1 AND Q2 ]
    remLines(inputFile, outputFile);


    printf("outputFile : %s\n", outputFile);
    // lexical analyzer
    lexicalAnalyzer(outputFile);
}
```

```c
#include <stdio.h>

int sum(int a, int b)
{
    int s=a+b;
    return s;
}

bool search(int *arr,int key) {
    int i;
    for(i=0;i<10;i++){
        if(arr[i]==key)
            return true;
        else return false;
    }
}


void main()
{
    int a[20],i,sum;
    bool status;
    printf("Enter array elements:");
    for(i=0;i<10;++i)
        scanf("%d",&a[i]);
    sum=a[0]+a[4];
    status=search(a,sum);
    printf("%d",status);
}
```

```
student@lplab-ThinkCentre-M71e: ~/190905494/CD/Week 4
student@lplab-ThinkCentre-M71e:~/190905494/CD/Week 4$ gcc 1_symTab.c -o q1.exe
student@lplab-ThinkCentre-M71e:~/190905494/CD/Week 4$ ./q1.exe
=== Our Symbol Table is ===
<s, int, 4>
<a, int, 4>
<b, int, 4>
<sum, func, 0>
<search, func, 0>
<i, int, 4>
<arr, int, 40>
<key, int, 4>
<Enter array elements:, char, 80>
student@lplab-ThinkCentre-M71e:~/190905494/CD/Week 4$
```