Week4

Name : NEERAJ KUMAR
Regno. : 220905536
Roll no. : 57
Title: CONSTRUCTION OF SYMBOL TABLE

Lab Exercise:

1. Using getNextToken() implemented in Lab No 3, design a Lexical Analyser to implement the following symbol tables. a. local symbol table

   Source Codes:

   a. Header file of Lab3(myfun.h):

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

#define MAX_TOKEN_LENGTH 100
#define MAX_TOKENS 1000

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;

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;
}

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;
}
```

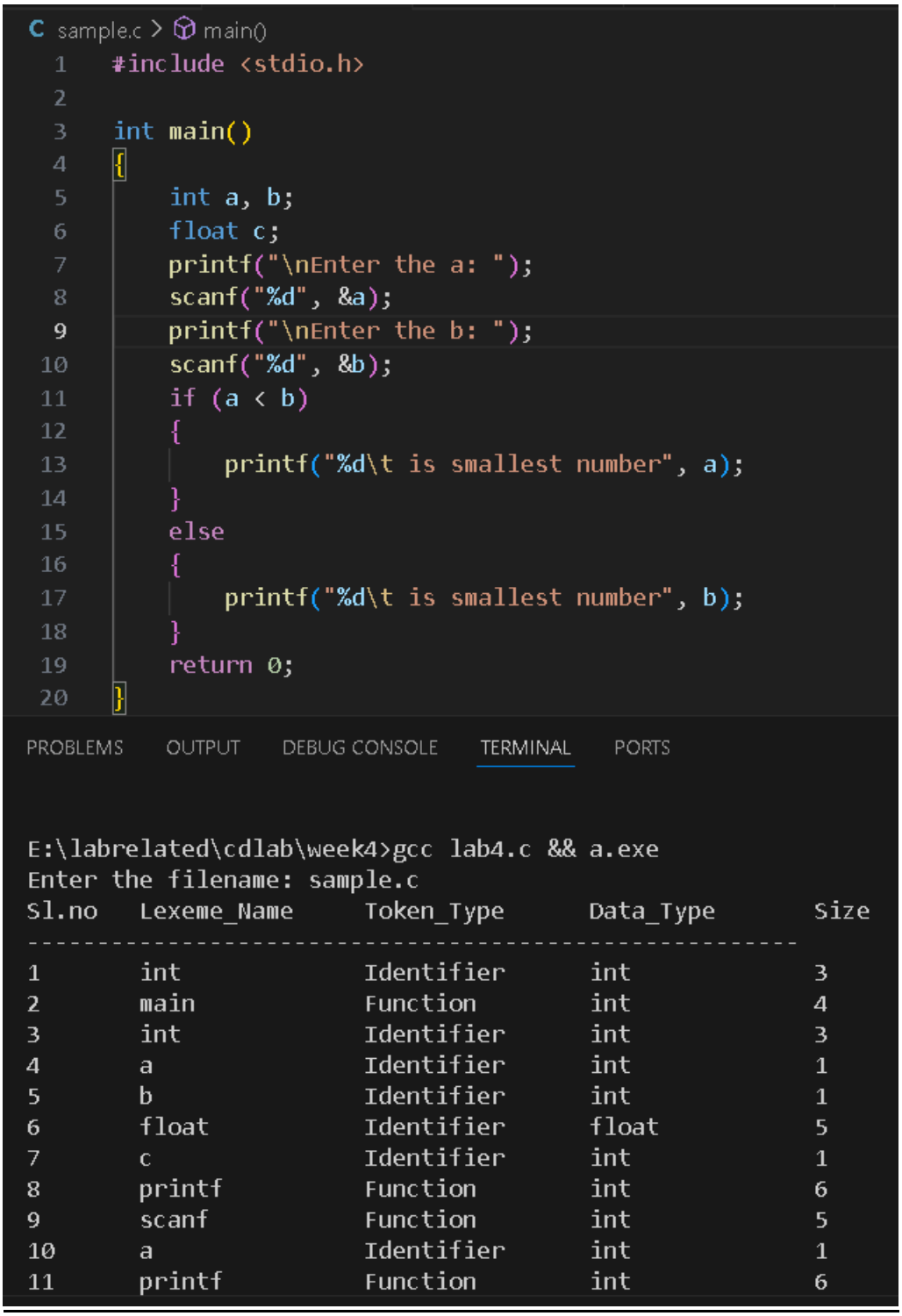
b. <u>Source Code of lab4:</u>

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

#define TABLE_SIZE 100  // Define the size of the hash table

// Define the symbol table entry structure
typedef struct {
```

```
    int slno;
    char lexemeName[128];
    char tokenType[128];
    char datatype[128];
    int size;
} Local;

// Define a hash table to store the local symbol table
Local symbolTable[TABLE_SIZE];

// Hash function to calculate the index
unsigned int hashFunction(char *str) {
    unsigned int hash = 5381;
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    }
    return hash % TABLE_SIZE;
}

int insertSymbolTable(Local symbolTable[], char *lexemeName, char *tokenType, char
*datatype, int size) {
    unsigned int index = hashFunction(lexemeName);

    for (int i = 0; i < TABLE_SIZE; i++) {
        if (symbolTable[i].lexemeName[0] == '\0') {
            symbolTable[i].slno = i + 1;
            strncpy(symbolTable[i].lexemeName, lexemeName, sizeof(symbolTable[i].lexemeName) -
1);
            strncpy(symbolTable[i].tokenType, tokenType, sizeof(symbolTable[i].tokenType) - 1);
            strncpy(symbolTable[i].datatype, datatype, sizeof(symbolTable[i].datatype) - 1);
            symbolTable[i].size = size;
            return 1;  // Insertion successful
        }
    }

    return 0;  // Symbol table is full
}

void displaySymbolTable(Local symbolTable[]) {
    printf("Sl.no\tLexeme_Name\tToken_Type\tData_Type\tSize\n");
    printf("-------------------------------------------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (symbolTable[i].lexemeName[0] != '\0') { // non-empty slot
            printf("%d\t%s\t\t%s\t%s\t\t%d\n", symbolTable[i].slno, symbolTable[i].lexemeName,
                symbolTable[i].tokenType, symbolTable[i].datatype, symbolTable[i].size);
        }
    }
}

// Function to determine if a string represents a basic data type
int isDataType(char *lexeme) {
```

```
    char *dataTypes[] = {"int", "float", "char", "double", "long", "short", "void"};
    for (int i = 0; i < sizeof(dataTypes) / sizeof(dataTypes[0]); i++) {
        if (strcmp(lexeme, dataTypes[i]) == 0) {
            return 1;  // It's a data type
        }
    }
    return 0;  // Not a data type
}

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

    FILE *source = fopen(filename, "r");
    if (!source) {
        printf("Error opening file\n");
        return 1;
    }

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

    while ((token = getNextToken(source, &row, &col)).type != token_EOF) {
        if (token.type == token_ID || token.type == token_KEYWORD) {
            char datatype[128] = "int";  // Default datatype for identifiers
            int size = strlen(token.value);  // Placeholder for size based on lexeme length

            // Check if the token is a data type keyword
            if (isDataType(token.value)) {
                strncpy(datatype, token.value, sizeof(datatype) - 1); // Set the correct datatype
            }

            // Check if the token is an identifier and if it's followed by '(' (indicating it's a function)
            int nextChar = fgetc(source);  // Look ahead to check for '('
            if (nextChar == '(') {
                insertSymbolTable(symbolTable, token.value, "Function", datatype, size);
            } else {
                // If it's not a function, treat it as a regular identifier (variable)
                ungetc(nextChar, source);  // Put back the character for further processing
                insertSymbolTable(symbolTable, token.value, "Identifier", datatype, size);
            }
        }
    }
    fclose(source);

    // Display the local symbol table
    displaySymbolTable(symbolTable);
    return 0;
}
```

Output:



```c
#include <stdio.h>

int main()
{
    int a, b;
    float c;
    printf("\nEnter the a: ");
    scanf("%d", &a);
    printf("\nEnter the b: ");
    scanf("%d", &b);
    if (a < b)
    {
        printf("%d\t is smallest number", a);
    }
    else
    {
        printf("%d\t is smallest number", b);
    }
    return 0;
}
```

```
E:\labrelated\cdlab\week4>gcc lab4.c && a.exe
Enter the filename: sample.c
Sl.no   Lexeme_Name     Token_Type      Data_Type       Size
---------------------------------------------------------
1       int             Identifier      int             3
2       main            Function        int             4
3       int             Identifier      int             3
4       a               Identifier      int             1
5       b               Identifier      int             1
6       float           Identifier      float           5
7       c               Identifier      int             1
8       printf          Function        int             6
9       scanf           Function        int             5
10      a               Identifier      int             1
11      printf          Function        int             6
```