**COMPILER DESIGN LAB**

**LAB 4: CONSTRUCTION OF SYMBOL TABLE**

**ADITYA AGARWAL 220905106 ROLLNO. 14**

**Q1 - Using getNextToken() implemented in Lab No 3, design a Lexical Analyser to implement the single symbol table using closed hashing.**

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

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

typedef struct {
   int row;
   int col;
   int type;
```

```
    char value[100];
} Token;

typedef struct {
    int index;
    char name[100];
    char type[20];
    int size;
} SymbolTableEntry;

SymbolTableEntry symbolTable[100];
int symbolTableSize = 0;

int isKeyword(char *lexeme) {
    const char *keywords[] = {"int", "char", "float", "if", "else", "return", "for", "while",
"void"};
    for (int i = 0; i < 9; i++) {
        if (strcmp(lexeme, keywords[i]) == 0) return 1;
    }
    return 0;
}

Token getNextToken(FILE *fp, int *row, int *col) {
    Token t = {-1, -1, 0, ""};
    char c, lexeme[100];
    int lexemeIndex = 0, inStringLiteral = 0, inComment = 0;
```

```
while ((c = fgetc(fp)) != EOF) {
    (*col)++;
    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;
}
// Skip preprocessor directives and comments (same as before)
// ... [unchanged code for preprocessor, comments, string literals, etc.] ...

// 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);
   t.row = *row;
   t.col = *col;
   t.type = isKeyword(lexeme) ? KEYWORD : IDENTIFIER;
```

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

// Handle other tokens (same as before)
// ... [unchanged code for numbers, operators, etc.] ...
// 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;
    strcpy(t.value, "EOF");
    return t;
}

void insertToken(char *name, char *type, int size) {
    if (symbolTableSize < 100) {
        symbolTable[symbolTableSize].index = symbolTableSize + 1;
        strcpy(symbolTable[symbolTableSize].name, name);
        strcpy(symbolTable[symbolTableSize].type, type);
        symbolTable[symbolTableSize].size = size;
        symbolTableSize++;
    }
}

void displaySymbolTable() {
    printf("Index\tName\tType\tSize\n-----\t----\t----\t----\n");
```

```
    for (int i = 0; i < symbolTableSize; i++) {
        printf("%d\t%s\t%s\t%d\n", symbolTable[i].index, symbolTable[i].name,
symbolTable[i].type, symbolTable[i].size);
    }
}

int getSizeForType(const char *type) {
    if (strcmp(type, "int") == 0) return 4;
    if (strcmp(type, "float") == 0) return 8;
    if (strcmp(type, "char") == 0) return 1;
    return 0;
}

int main() {
    FILE *fp = fopen("sample.c", "r");
    if (!fp) return 1;

    int row = 1, col = 1;
    Token t;
    char currentType[20] = "";

    while ((t = getNextToken(fp, &row, &col)).type != 0) {
        if (t.type == KEYWORD) {
            // Check if the keyword is a type (int, float, char)
            if (strcmp(t.value, "int") == 0 || strcmp(t.value, "float") == 0 || strcmp(t.value, "char")
== 0) {
                strcpy(currentType, t.value);
```

```
            }
        } else if (t.type == IDENTIFIER && strcmp(currentType, "") != 0) {
            // Check if the identifier is part of a function declaration
            char nextChar = fgetc(fp);
            ungetc(nextChar, fp); // Peek ahead without consuming

            if (nextChar != '(') { // Not a function; add to symbol table
                int size = getSizeForType(currentType);
                insertToken(t.value, currentType, size);
            }
            strcpy(currentType, ""); // Reset type
        }
    }

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

SAMPLE –

```
#include <stdio.h>

#define PI 3.14  // Constant definition
```

```
// Structure definition
struct Student {
    int id;
    char name[50];
    float marks;
};

// Function prototype
int add(int a, int b);

int main() {
    int count = 5;
    float temperature = 36.5;
    char grade = 'A';

    // Array declaration
    int numbers[5] = {1, 2, 3, 4, 5};

    // Pointer declaration
    int *ptr = &count;

    // Loop example
    for(int i = 0; i < count; i++) {
        printf("Number: %d\n", numbers[i]);
    }
```

```
    // Conditional example
    if(temperature > 37.0) {
        printf("Fever detected!\n");
    } else {
        printf("Normal temperature.\n");
    }

    // Function call
    int result = add(count, 10);

    // Structure usage
    struct Student s1;
    s1.id = 101;
    s1.marks = 95.5;

    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b;
}
```

OUTPUT:

```
// Output Format:
// Index   Name          Type    Size
// -----   ----          ----    ----
// 1       id            int     4
// 2       name          char    1
// 3       marks         float   8
// 4       a             int     4
// 5       b             int     4
// 6       count         int     4
// 7       temperature   float   8
// 8       grade         char    1
// 9       numbers       int     4
// 10      ptr           int     4
// 11      i             int     4
// 12      result        int     4
// 13      a             int     4
// 14      b             int     4
```