WPL LAB 4

name: Abhinav Singh Bhagtana
reg no: 230905225
roll no: A-029

Q1.Using getNextToken( ) implemented in Lab No 3, design a Lexical Analyser to implement the
following symbol tables.
a. local symbol table

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

#define TABLE_LENGTH 30

int ROW = 1, COL = 1;
FILE *f1;

typedef struct
{
   char token_name[100];
   unsigned int row, col;
   char type[50];
} Token;

typedef struct listElement
{
   Token *tok;
   struct listElement *next;
} listElement;

listElement *TABLE[TABLE_LENGTH] = {NULL};

// Check if a string is a data type keyword
int isDataType(char *str)
{
   if (strcmp(str, "int") == 0 || strcmp(str, "double") == 0 ||
      strcmp(str, "char") == 0 || strcmp(str, "float") == 0 ||
      strcmp(str, "bool") == 0)
   {
      return 1;
   }
   return 0;
}

// Check for other keywords
```

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

int generateHash(char *name)
{
    int sum = 0;
    for (int i = 0; name[i] != '\0'; i++)
        sum += (int)name[i];
    return sum % TABLE_LENGTH;
}

// Prevent duplicates and find existing tokens
Token *search(char *name)
{
    int hash = generateHash(name);
    listElement *curr = TABLE[hash];
    while (curr)
    {
        if (strcmp(curr->tok->token_name, name) == 0)
            return curr->tok;
        curr = curr->next;
    }
    return NULL;
}

void insert(Token t)
{
    // Only insert identifiers into the symbol table
    if (strcmp(t.type, "id") != 0 && !isDataType(t.type))
        return;

    // Don't insert if it already exists
    if (search(t.token_name))
        return;

    int hash = generateHash(t.token_name);
    listElement *newNode = (listElement *)malloc(sizeof(listElement));
```

```
    // Create a permanent copy of the token
    newNode->tok = (Token *)malloc(sizeof(Token));
    *(newNode->tok) = t;
    newNode->next = TABLE[hash]; // Insert at head for simplicity
    TABLE[hash] = newNode;
}

void displaySymTable()
{
    printf("\n--- SYMBOL TABLE ---\n");
    printf("%-10s %-15s %-10s %-5s\n", "HashIdx", "Lexeme", "Type", "Size");
    for (int i = 0; i < TABLE_LENGTH; i++)
    {
        listElement *curr = TABLE[i];
        while (curr != NULL)
        {
            int size = 0;
            if (strcmp(curr->tok->type, "int") == 0)
                size = 4;
            else if (strcmp(curr->tok->type, "double") == 0)
                size = 8;
            else if (strcmp(curr->tok->type, "char") == 0)
                size = 1;
            else
                size = 0;

            printf("%-10d %-15s %-10s %-5d\n", i, curr->tok->token_name, curr->tok->type,
size);
            curr = curr->next;
        }
    }
}

Token getNextToken()
{
    Token t;
    t.token_name[0] = '\0';
    int c;

    while ((c = fgetc(f1)) != EOF)
    {
        // 1. Skip Whitespace & Update Line/Col
        if (c == '\n')
        {
            ROW++;
            COL = 1;
            continue;
        }
```

```
if (isspace(c))
{
   COL++;
   continue;
}

// 2. Skip Preprocessor Directives
if (c == '#')
{
   while ((c = fgetc(f1)) != '\n' && c != EOF)
      ;
   ROW++;
   COL = 1;
   continue;
}

// 3. Skip Comments (Single and Multi-line)
if (c == '/')
{
   int next = fgetc(f1);
   if (next == '/')
   { // Single line
      while ((c = fgetc(f1)) != '\n' && c != EOF)
         ;
      ROW++;
      COL = 1;
      continue;
   }
   else if (next == '*')
   { // multiline
      while (1)
      {
         c = fgetc(f1);
         if (c == EOF)
            break;
         if (c == '\n')
         {
            ROW++;
            COL = 1;
         }
         else
            COL++;

         if (c == '*')
         {
            int postStar = fgetc(f1);
            if (postStar == '/')
            {
```

```
                COL++;
                break;
            } // Found end
            else
                ungetc(postStar, f1); // Not the end, put it back
        }
    }
    continue;
  }
  else
    ungetc(next, f1); // Just a division operator
}

// --- Start of Token Identification ---
t.row = ROW;
t.col = COL;

// 4. Identifiers and Keywords
if (isalpha(c) || c == '_')
{
  int i = 0;
  t.token_name[i++] = c;
  while (isalnum(c = fgetc(f1)) || c == '_')
  {
    if (i < 99)
      t.token_name[i++] = c;
  }
  ungetc(c, f1);
  t.token_name[i] = '\0';
  COL += i;
  if (isKeyword(t.token_name))
    strcpy(t.type, t.token_name);
  else
    strcpy(t.type, "id");
  return t;
}

// 5. Numerical Constants (handles decimal)
if (isdigit(c))
{
  int i = 0;
  t.token_name[i++] = c;
  int hasDecimal = 0;
  while (1)
  {
    c = fgetc(f1);
    if (isdigit(c))
    {
```

```
            if (i < 99)
                t.token_name[i++] = c;
        }
        else if (c == '.' && !hasDecimal)
        {
            hasDecimal = 1;
            if (i < 99)
                t.token_name[i++] = c;
        }
        else
        {
            ungetc(c, f1);
            break;
        }
    }
    t.token_name[i] = '\0';
    COL += i;
    strcpy(t.type, "num");
    return t;
}

// 6. String Literals
if (c == '"')
{
    int i = 0;
    while ((c = fgetc(f1)) != '"' && c != EOF)
    {
        if (c == '\n')
        {
            ROW++;
            COL = 1;
        }
        if (i < 99)
            t.token_name[i++] = c;
    }
    t.token_name[i] = '\0';
    COL += (i + 2);
    strcpy(t.type, "string");
    return t;
}

// 7. Arithmetic, Relational, and Logical Operators
t.token_name[0] = c;
t.token_name[1] = '\0';
if (strchr("+-*/%=!><&|", c))
{
    int next = fgetc(f1);
    // Check for compound: ==, !=, <=, >=, ++, --, &&, ||
```

```
            if ((next == '=') ||
                (c == '+' && next == '+') ||
                (c == '-' && next == '-') ||
                (c == '&' && next == '&') ||
                (c == '|' && next == '|'))
            {
                t.token_name[1] = next;
                t.token_name[2] = '\0';
                COL += 2;
            }
            else
            {
                ungetc(next, f1);
                COL += 1;
            }
            strcpy(t.type, t.token_name);
            return t;
        }

        // 8. Special Symbols (parentheses, brackets, commas, etc.)
        if (strchr("(){}[],.;", c))
        {
            strcpy(t.type, t.token_name);
            COL++;
            return t;
        }

        // Unknown character
        strcpy(t.type, "unknown");
        COL++;
        return t;
    }
    t.row = 0; // EOF Signal
    return t;
}

int main()
{
    f1 = fopen("input.c", "r");
    if (!f1)
    {
        printf("Error opening file\n");
        return 1;
    }

    Token t;
    char currentContextType[50] = "unknown";
```

```
    while (1)
    {
        t = getNextToken();
        if (t.row == 0)
            break;

        // LOGIC: If we see "int", remember it for the next "id"
        if (isDataType(t.token_name))
        {
            strcpy(currentContextType, t.token_name);
        }
        else if (strcmp(t.type, "id") == 0)
        {
            // Apply the remembered type to the identifier
            strcpy(t.type, currentContextType);
            insert(t);
        }

        // Reset type after a semicolon (end of statement)
        if (strcmp(t.token_name, ";") == 0)
        {
            strcpy(currentContextType, "unknown");
        }
    }

    displaySymTable();
    fclose(f1);
    return 0;
}
```

input:

```
#include <stdio.h>
//#include <stdio.h>
/*
this is multiline comment
*/
int main()
{
int a,b,sum;
int ab,ba;
double abc;
a = 1;
b = 1;
sum = a+b;
printf("hlo");
}
```

output:

```
--- SYMBOL TABLE ---
HashIdx    Lexeme          Type       Size
```

1	main	int	4
7	a	int	4
8	b	int	4
11	sum	int	4
15	ba	int	4
15	ab	int	4
24	abc	double	8