SSN COLLEGE OF ENGINEERING (Autonomous)
Affiliated to Anna University
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject: **Compiler Design Lab** Acad. Year: 2024-2025 (Odd)

Class: VII Sem - IV-year Roll No: 3122215001085

Sub. Code: **UCS2702** Name: Rohith M

Batch: 2021-2025
Staff In charge: Dr. B Bharathi
Date: 11-08-2024

Assignment-1 - Implementation of lexical analyser and symbol table

Question:

The Goal

In the first programming assignment, you will get your compiler off to a great start by implementing Lexical analyzer or Scanner using C. Your scanner will run through the source program recognizing C tokens in the order in which they are read, until end of file is reached. When an identifier is encountered, it should be stored in symbol table with its attributes. Symbol table consist of the attributes identifier name, type, no of bytes, location and value. Your scanner should identify the tokens categorized below and print it.

C Programming Language: Lexical construct

Here is the summary of the token types in C programming language

The following are **keywords**. They are all reserved:

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

An **identifier** is a sequence of letters, digits and underscores, starting with a letter. C language is case sensitive. Eg; “if” is a keyword, but “IF” is an identifier. Binky and binky are two different identifiers.

C language adopts two types of **comments**. A single-line comment is started by // and extends to the end of the line. Multi-line comments start with /* and end with the first subsequent */. Multi-line comments do not nest. If a file ends with an unterminated comment, the scanner should report an error.

An integer **constant** can either be specified in decimal (base 10) or hexadecimal (base 16). A decimal integer is a sequence of decimal digits (0-9). A double constant is a sequence of digits, a period, followed by any sequence of digits. A string constant is a sequence of characters enclosed in double quotes. Strings can contain any character except a new line or double quote. A string must start and end on a single line; it cannot be split over multiple lines:

Arithmetic **operators**

+, -, *, /, %

Arithmetic assignment operators

+=, -=, *=, /=, %=

Logical operators

&&, ||, !

Relational operators

<=, >, >=, ==, !=

Bitwise operators

^, &, |, <>

Unary operators

-, ++, - -

Assignment operator

=

Special character

; , . [ ] ( ) { } []

The following are identified as function calls

printf(), scanf(), getch(), clrscr(),

AIM:

Develop a scanner that will recognize all the above specified tokens. Test your program for all specified tokens.

CODE (The Scanner):

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

// Token types
typedef enum {
    KEYWORD, IDENTIFIER, INTEGER_CONSTANT, DOUBLE_CONSTANT,
    STRING_CONSTANT, ARITHMETIC_OPERATOR, LOGICAL_OPERATOR,
    RELATIONAL_OPERATOR, BITWISE_OPERATOR, UNARY_OPERATOR,
    ASSIGNMENT_OPERATOR, SPECIAL_CHARACTER, FUNCTION_CALL,
PREPROCESSOR_DIRECTIVE
} TokenType;

// Keywords list
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"
};

// Function calls list
const char *functionCalls[] = {
    "printf", "scanf", "getch", "clrscr"
};

// Symbol table entry
typedef struct {
    char name[50];
    char type[10];
    int  bytes;
    int address;
    int value;
} SymbolTableEntry;

SymbolTableEntry symbolTable[100];
int symbolTableIndex = 0;

// Check if the given string is a keyword
int isKeyword(char *str) {
    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Check if the given string is a function call
int isFunctionCall(char *str) {
    for (int i = 0; i < sizeof(functionCalls) / sizeof(functionCalls[0]); i++)
{
        if (strcmp(str, functionCalls[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Add an identifier to the symbol table
void addToSymbolTable(char *name, char *type, int bytes, int address, int
value) {
    // Check for duplicates
    for (int i = 0; i < symbolTableIndex; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            // Update the value if the identifier already exists
            symbolTable[i].value = value;
            return;
        }
    }
    strcpy(symbolTable[symbolTableIndex].name, name);
    strcpy(symbolTable[symbolTableIndex].type, type);
    symbolTable[symbolTableIndex].bytes = bytes;
    symbolTable[symbolTableIndex].address = address;
    symbolTable[symbolTableIndex].value = value;
    symbolTableIndex++;
}

// Find identifier in the symbol table
int findIdentifierIndex(char *name) {
    for (int i = 0; i < symbolTableIndex; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return i;
        }
    }
    return -1;
}

void scan(char *sourceCode) {
    char token[100];
    int i = 0, j = 0, address = 1000;
    char currentIdentifier[50] = {0}; // Track the latest identifier

    while (sourceCode[i] != '\0') {
        // Skip white spaces
        if (isspace(sourceCode[i])) {
            i++;
            continue;
        }

        // Identify preprocessor directives
        if (sourceCode[i] == '#') {
            j = 0;
            while (sourceCode[i] != '\n' && sourceCode[i] != '\0') {
                token[j++] = sourceCode[i++];
            }
            token[j] = '\0';
            printf("%s - preprocessor directive\n", token);
            continue;
        }

        // Identify keywords, identifiers, and function calls
        if (isalpha(sourceCode[i])) {
            j = 0;
            while (isalnum(sourceCode[i]) || sourceCode[i] == '_') {
                token[j++] = sourceCode[i++];
            }
            token[j] = '\0';

            if (isKeyword(token)) {
                printf("%s - keyword\n", token);
            } else if (sourceCode[i] == '(') {
                // Read the entire function call
                printf("%s(", token);
                i++;
                while (sourceCode[i] != ')' && sourceCode[i] != '\0') {
                    printf("%c", sourceCode[i++]);
                }
                if (sourceCode[i] == ')') {
                    printf(") - function call\n");
                    i++;
                }
            } else {
                printf("%s - identifier\n", token);
                // Store the current identifier
                strcpy(currentIdentifier, token);
                if (findIdentifierIndex(token) == -1) {
                    addToSymbolTable(token, "int", 2, address, 0); // Assuming
type int and initial value 0 for simplicity
```

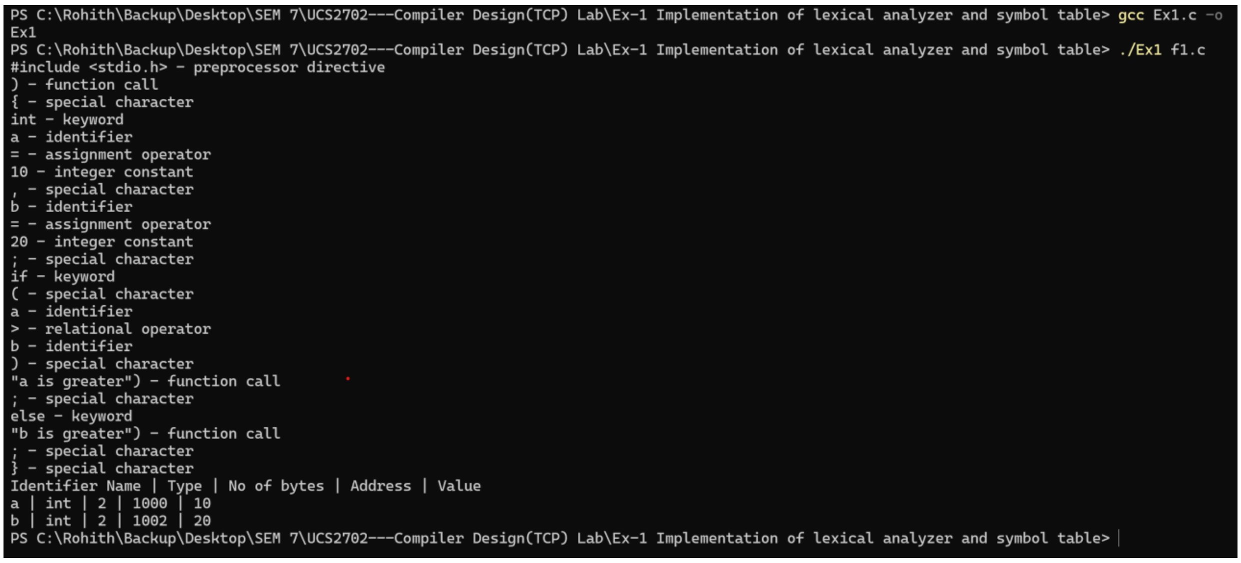
```c
                    address += 2;
                }
            }
            continue;
        }

        // Identify integer and double constants
        if (isdigit(sourceCode[i])) {
            j = 0;
            while (isdigit(sourceCode[i])) {
                token[j++] = sourceCode[i++];
            }
            token[j] = '\0';

            if (sourceCode[i] == '.') {
                token[j++] = sourceCode[i++];
                while (isdigit(sourceCode[i])) {
                    token[j++] = sourceCode[i++];
                }
                token[j] = '\0';
                printf("%s - double constant\n", token);
            } else {
                printf("%s - integer constant\n", token);
            }
            continue;
        }

        // Identify string constants
        if (sourceCode[i] == '"') {
            j = 0;
            token[j++] = sourceCode[i++];
            while (sourceCode[i] != '"' && sourceCode[i] != '\0') {
                token[j++] = sourceCode[i++];
            }
            if (sourceCode[i] == '"') {
                token[j++] = sourceCode[i++];
                token[j] = '\0';
                printf("%s - string constant\n", token);
            } else {
                printf("Error: Unterminated string constant\n");
            }
            continue;
        }

        // Identify operators and special characters
        switch (sourceCode[i]) {
            case '+': case '-': case '*': case '/': case '%':
                printf("%c - arithmetic operator\n", sourceCode[i]);
                i++;
                break;
            case '=':
                if (sourceCode[i + 1] == '=') {
                    printf("== - relational operator\n");
                    i += 2;
                } else {
                    printf("= - assignment operator\n");
                    i++;
                    // Handle assignment to identifier
                    while (isspace(sourceCode[i])) {
                        i++;
                    }
                    if (isdigit(sourceCode[i])) {
                        j = 0;
                        while (isdigit(sourceCode[i])) {
                            token[j++] = sourceCode[i++];
                        }
                        token[j] = '\0';
                        int value = atoi(token);
                        printf("%s - integer constant\n", token); // Print the
integer constant
                        // Update the value of the current identifier
                        if (strlen(currentIdentifier) > 0) {
                            int index =
findIdentifierIndex(currentIdentifier);
                            if (index != -1) {
                                symbolTable[index].value = value;
                            }
                        }
                    }
                }
                break;
            case '!':
                if (sourceCode[i + 1] == '=') {
                    printf("!= - relational operator\n");
                    i += 2;
                } else {
                    printf("! - logical operator\n");
                    i++;
                }
                break;
            case '<': case '>':
                if (sourceCode[i + 1] == '=') {
                    printf("%c= - relational operator\n", sourceCode[i]);
                    i += 2;
                } else {
                    printf("%c - relational operator\n", sourceCode[i]);
```

```c
                    i++;
                }
                break;
            case '&': case '|':
                if (sourceCode[i] == sourceCode[i + 1]) {
                    printf("%c%c - logical operator\n", sourceCode[i],
sourceCode[i]);
                    i += 2;
                } else {
                    printf("%c - bitwise operator\n", sourceCode[i]);
                    i++;
                }
                break;
            case '^': case '~': case '(':
            case ')': case '{': case '}': case '[': case ']':
            case ',': case ';': case '.':
                printf("%c - special character\n", sourceCode[i]);
                i++;
                break;
            default:
                printf("Unknown character: %c\n", sourceCode[i]);
                i++;
                break;
        }
    }
}


int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // Open the file specified by the command line argument
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Determine the size of the file
    fseek(file, 0, SEEK_END);
    long fileSize = ftell(file);
    fseek(file, 0, SEEK_SET);

    // Allocate memory for the source code
    char *sourceCode = malloc(fileSize + 1);
    if (sourceCode == NULL) {
        perror("Error allocating memory");
        fclose(file);
        return 1;
    }

    // Read the file content into sourceCode
    size_t bytesRead = fread(sourceCode, 1, fileSize, file);
    if (bytesRead != fileSize) {
        perror("Error reading file");
        free(sourceCode);
        fclose(file);
        return 1;
    }
    sourceCode[bytesRead] = '\0'; // Null-terminate the string

    // Close the file
    fclose(file);

    // Scan the source code
    scan(sourceCode);

    // Free the allocated memory
    free(sourceCode);

    // Print the symbol table
    printf("\nContent of Symbol Table:\n");
    printf("Identifier Name | Type | No of bytes | Address | Value\n");
    for (int i = 0; i < symbolTableIndex; i++) {
        printf("%s | %s | %d | %d | %d\n",
               symbolTable[i].name, symbolTable[i].type,
               symbolTable[i].bytes, symbolTable[i].address,
symbolTable[i].value);
    }

    return 0;
}
```

Input program to be scanned (filename given as command line argument)

```
#include<stdio.h>
main()
{
int a=10,b=20;
if(a>b)
printf("a is greater");
else
printf("b is greater");
}
```

OUTPUT:



```
PS C:\Rohith\Backup\Desktop\SEM 7\UCS2702---Compiler Design(TCP) Lab\Ex-1 Implementation of lexical analyzer and symbol table> gcc Ex1.c -o
Ex1
PS C:\Rohith\Backup\Desktop\SEM 7\UCS2702---Compiler Design(TCP) Lab\Ex-1 Implementation of lexical analyzer and symbol table> ./Ex1 f1.c
#include <stdio.h> - preprocessor directive
) - function call
{ - special character
int - keyword
a - identifier
= - assignment operator
10 - integer constant
, - special character
b - identifier
= - assignment operator
20 - integer constant
; - special character
if - keyword
( - special character
a - identifier
> - relational operator
b - identifier
) - special character
"a is greater") - function call
; - special character
else - keyword
"b is greater") - function call
; - special character
} - special character
Identifier Name | Type | No of bytes | Address | Value
a | int | 2 | 1000 | 10
b | int | 2 | 1002 | 20
PS C:\Rohith\Backup\Desktop\SEM 7\UCS2702---Compiler Design(TCP) Lab\Ex-1 Implementation of lexical analyzer and symbol table>
```

LEARNING OUTCOME:

- Learn lexical analysis by implementing a scanner for C tokens
- Identify and categorize different types of C tokens
- Manage a symbol table with identifier attributes
- Handle string constants and comments, including error checking
- Recognize and differentiate function calls from identifiers