L. D. College of Engineering

Opp Gujarat University, Navrangpura, Ahmedabad - 380015

LAB MANUAL

Branch: Computer Engineering

CD – COMPILER DESIGN (3170701)

Semester: VII

Faculty Details:

1) Prof. Pinal D. Salot
2) Prof. Prachi V. Pancholi

Certificate

This is to certify that Shri/Ms. **UJAS TULSIBHAI LATHIYA** ***Enrollmement No*** **210280107052** ***of BE Sem*** **7 th** ***class has Satisfactorily completed the coursein*** **COMPILER DESIGN** ***within four walls of L. D. College of Engineering, Ahmedabad – 380015.***

Date of Submission :- ____

Staff in – Charge :- ____

Head of Department :- ____

INDEX

9	**5**	Write a program to implement LALR parsing.				
10	**5**	Write a program to implement operator precedence parsing.				
11	**5**	To Study about Yet Another Compiler-Compiler(YACC) and Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .				
12	**5**	Implement 3- address code(any one) w.r.t intermediate code for given infix expression.				

L. D. College of Engineering, Ahmedabad

Department of Computer Engineering

Subject Name: Compiler Design **Subject Code:3170701**

Term: 2024-25

Rubrics ID	Criteria	Marks	Good (2)	Satisfactory (1)	Need Improvement (0)
RB1	Regularity	05	High (>70%)	Moderate (40-70%)	Poor (0-40%)
RB2	Problem Analysis & Development of the Solution	05	Apt & Full Identification of the Problem& Complete Solution for the Problem	Limited Identification of the Problem / Incomplete Solution for the Problem	Very Less Identification of the Problem / Very Less Solution for the Problem
RB3	Practical Output	05	Correct output as required	Partially Correct Output for the Problem	No output or Output with error(s)
RB4	Mock viva test	05	All questions responded Correctly	Delayed & partially correct response	Very few questions answered correctly

SIGN OF FACULTY

L. D. College of Engineering, Ahmedabad

Department of Computer Engineering

LABORATORY PRACTICALS ASSESSMENT

Subject Name: COMPILER DESIGN **Subject Code:3170701**

Term: 2024-25

Pract. No.	CO No.	RB1	RB2	RB3	RB4	Total	Date	Faculty Sign
1	5							
2	5							
3	5							
4	5							
5	5							
6	5							
7	5							
8	5							
9	5							
10	5							
11	5							
12	5							

GUJARAT TECHNOLOGICAL UNIVERSITY, AHMEDABAD,

COURSE CURRICULUM

COURSE TITLE: COMPILER DESIGN (3170701)

(Code: 3170701)

Degree Programs in which this course is offered	Semester in which offered
Computer Engineering	7th Semester

1. RATIONALE

- Compiler Design is a fundamental subject of Computer Engineering
- Compiler design principles provide an in-depth view of translation, optimization and compilation of the entire source program.
- It also focuses on various designs of compiler and structuring of various phases of compiler.
- It is inevitable to grasp the knowledge of various types of grammar, lexical analysis, yacc, FSM(Finite State Machines) and correlative concepts of languages.

2. COMPETENCY

The course content should be taught and analyze with the aim to develop different types of skills so that students are able to acquire following competency:

Compiler Design subject is needed to grasp knowledge of various language grammar and concepts.

3. COURSE OUTCOMES

After learning the course, the students should be able to:

1. Explain the basic concepts and application of Compiler Design.
2. A Use lexical analysis techniques.
3. Apply parsing & semantic analysis techniques.
4. Understand issues of run time environments, code generation and optimization techniques.

5. Design a simple tiny compiler.

4. TEACHING AND EXAMINATION SCHEME

Teaching Scheme			Credits	Examination Marks				Total Marks
L	T	P	C	Theory Marks		Practical Marks		
				ESE (E)	PA (M)	ESE (V)	PA (I)	
3	0	2	4	70	30	30	20	150

5. SUGGESTED LEARNING RESOURCES

A. LIST OF BOOKS

1. Compiler Tools Techniques - A.V.Aho, Ravi Sethi, J.D.Ullman, Addison Wesley
2. The Theory And Practice Of Compiler Writing - Trembley J.P. And Sorenson P.G. Mcgraw-Hill
3. Modern Compiler Design - Dick Grune, Henri E. Bal, Jacob, Langendoen, WILEY India
4. Compiler Construction-Principles And Practices - D.M.Dhamdhere, Mcmillian
5. Principles of Compiler Design, V. Raghavan, McGrawHill
6. Compiler Construction - Waite W.N. And Goos G., Springer Verlag

B. LIST OF SOFTWARE / LEARNING WEBSITES

- https://nptel.ac.in/courses/106/105/106105190/

Importance of Compiler Design Lab

Compiler is a software which takes as input a program written in a High-Level language and translates it into its equivalent program in Low Level program. Compilers teaches us how real-world applications are working and how to design them.

Learning Compilers gives us with both theoretical and practical knowledge that is crucial in order to implement a programming language. It gives you a new level of understanding of a language in order to make better use of the language (optimization is just one example). Sometimes just using a compiler is not enough. You need to optimize the compiler itself for your application.

Compilers have a general structure that can be applied in many other applications, from debuggers to simulators to 3D applications to a browser and even a cmd / shell. understanding compilers and how they work makes it super simple to understand all the rest. a bit like a deep understanding of math will help you to understand geometry or physics. We cannot do physics without the math. not on the same level. Just using something (read: tool, device, software, programming language) is usually enough when everything goes as expected. But if something goes wrong, only a true understanding of the inner workings and details will help to fix it. Even more specifically, Compilers are super elaborated / sophisticated systems (architecturally speaking). If you will say that can or have written a compiler by yourself - there will be no doubt as to your capabilities as a programmer. There is nothing you cannot do in the Software realm. So, better be a pilot who have the knowledge and mechanics of an airplane than the one who just know how to fly.

Every computer scientist can do much better if have knowledge of compilers apart from the domain and technical knowledge. Compiler design lab provides deep understanding of how programming language Syntax, Semantics are used in translation into machine equivalents apart from the knowledge of various compiler generation tools like LEX,YACC etc.

Practical – 1

Aim: Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.

❖ **Definition:**

- To design a lexical analyzer (also known as a lexer or scanner) in C that ignores redundant spaces, tabs, new lines, and comments, we can follow these steps:

  1. **Define the Tokens**: Identify the tokens we want to recognize (keywords, identifiers, operators, etc.).
  2. **Read Input**: Read the source code as input.
  3. **Skip Whitespaces and Comments**: Ignore spaces, tabs, new lines, and comments.
  4. **Recognize Tokens**: Use regular expressions or simple state machines to identify tokens.
  5. **Output Tokens**: Print recognized tokens and their types.

❖ Implementation:

- **Example Implementation of a Lexical Analyzer in C:**
  - ➢ Here is a simple implementation of a lexical analyzer in C. This example will recognize keywords, identifiers, numbers, and operators while ignoring whitespace and comments. The length of identifiers is restricted to a reasonable value (e.g., 32 characters).

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

#define MAX_IDENTIFIER_LENGTH 32

typedef enum {
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_KEYWORD,
    TOKEN_OPERATOR,
    TOKEN_EOF,
    TOKEN_UNKNOWN
} TokenType;

const char* keywords[] = {"int", "float", "if", "else",
"return", "while", NULL};

typedef struct {
```

```
    TokenType type;
    char value[MAX_IDENTIFIER_LENGTH + 1];
} Token;

int is_keyword(const char* identifier) {
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(keywords[i], identifier) == 0) {
            return 1;
        }
    }
    return 0;
}

Token get_next_token(FILE* fp) {
    Token token;
    char ch;
    int index = 0;

    // Skip whitespace
    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) {
            continue;  // Ignore whitespace
        } else if (ch == '/') {
            // Check for comment
            if ((ch = fgetc(fp)) == '/') {
                while ((ch = fgetc(fp)) != EOF && ch !=
'\n'); // Ignore single-line comment
            } else if (ch == '*') {
```

```
                // Ignore multi-line comment
                while (1) {
                    ch = fgetc(fp);
                    if (ch == EOF) break;
                    if (ch == '*') {
                        if ((ch = fgetc(fp)) == '/')
break; // End of comment
                    }
                }
            } else {
                ungetc(ch, fp);  // Not a comment, put
it back
                break;
            }
        } else {
            break;  // Found a non-whitespace character
        }
    }

    // End of file
    if (ch == EOF) {
        token.type = TOKEN_EOF;
        strcpy(token.value, "EOF");
        return token;
    }

    // Identifiers and keywords
    if (isalpha(ch)) {
```

```
        token.type = TOKEN_IDENTIFIER;
        do {
            if (index < MAX_IDENTIFIER_LENGTH) {
                token.value[index++] = ch;
            }
            ch = fgetc(fp);
        } while (isalnum(ch) || ch == '_');
        ungetc(ch, fp); // Put back the non-identifier
character
        token.value[index] = '\0';
        if (is_keyword(token.value)) {
            token.type = TOKEN_KEYWORD;
        }
        return token;
    }

    // Numbers
    if (isdigit(ch)) {
        token.type = TOKEN_NUMBER;
        do {
            if (index < MAX_IDENTIFIER_LENGTH) {
                token.value[index++] = ch;
            }
            ch = fgetc(fp);
        } while (isdigit(ch));
        ungetc(ch, fp); // Put back the non-number
character
        token.value[index] = '\0';
```

```
        return token;
    }

    // Operators
    token.type = TOKEN_OPERATOR;
    token.value[0] = ch;
    token.value[1] = '\0';
    return token;
}

int main() {
    FILE* fp = fopen("source_code.txt", "r");
    if (fp == NULL) {
        fprintf(stderr, "Could not open file.\n");
        return 1;
    }

    Token token;
    do {
        token = get_next_token(fp);
        switch (token.type) {
            case TOKEN_IDENTIFIER:
                printf("IDENTIFIER: %s\n",
token.value);
                break;
            case TOKEN_NUMBER:
                printf("NUMBER: %s\n", token.value);
                break;
```

```
                case TOKEN_KEYWORD:
                    printf("KEYWORD: %s\n", token.value);
                    break;
                case TOKEN_OPERATOR:
                    printf("OPERATOR: %s\n", token.value);
                    break;
                case TOKEN_EOF:
                    printf("EOF reached\n");
                    break;
                default:
                    printf("UNKNOWN TOKEN: %s\n",
token.value);
                    break;
            }
    } while (token.type != TOKEN_EOF);

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

- **Sample Input File (source_code.txt):**

```
// This is a comment
int main() {
    float value = 10.5; // Another comment
    if (value > 10) {
        return 1;
    } else {
        return 0;
    }
}

/* Multi-line
   comment */
```

- **Output:**

```
KEYWORD: int
IDENTIFIER: main
OPERATOR: (
OPERATOR: )
OPERATOR: {
KEYWORD: float
IDENTIFIER: value
OPERATOR: =
NUMBER: 10
OPERATOR: .
NUMBER: 5
IDENTIFIER: if
```

```
OPERATOR: (
IDENTIFIER: value
OPERATOR: >
NUMBER: 10
OPERATOR: )
OPERATOR: {
KEYWORD: return
NUMBER: 1
OPERATOR: ;
OPERATOR: }
KEYWORD: else
OPERATOR: {
KEYWORD: return
NUMBER: 0
OPERATOR: ;
OPERATOR: }
OPERATOR: }
EOF reached
```

Practical – 2

Aim: Write a C program to identify whether a given line is a comment or not.

- **Description:**
  - To create a C program that identifies whether a given line is a comment, we can check for two types of comments:
    1. **Single-line comments**: These start with // and continue until the end of the line.
    2. **Multi-line comments**: These start with /* and end with */.

- **Implementation:**
  - **C Program to Identify Comments:**

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

#define MAX_LINE_LENGTH 256

bool is_single_line_comment(const char *line) {
    // Check for single-line comment
    char *comment_start = strstr(line, "//");
    return comment_start != NULL && comment_start[2]
== '\0';
}

bool is_multi_line_comment(const char *line) {
```

```
    // Check for multi-line comment
    return strstr(line, "/*") != NULL &&
strstr(line, "*/") != NULL;
}

bool is_comment(const char *line) {
    // Check if the line is a comment
    return is_single_line_comment(line) ||
is_multi_line_comment(line);
}

int main() {
    char line[MAX_LINE_LENGTH];

    printf("Enter a line of code: ");
    fgets(line, sizeof(line), stdin); // Read a
line from standard input

    // Remove newline character from the end
    line[strcspn(line, "\n")] = '\0';

    if (is_comment(line)) {
        printf("The line is a comment.\n");
    } else {
        printf("The line is not a comment.\n");
    }

    return 0;
}
```

- **Output:**

```
Enter a line of code: // This is a
comment
The line is a comment.
Enter a line of code: int main() {
return 0; }
The line is not a comment.
```

Practical – 3

Aim: Write a C program to test whether a given identifier is valid or not.

❖ **Description:**

- To create a C program that tests whether a given identifier is valid according to the rules of C programming, we need to consider the following criteria for a valid identifier:

1. An identifier must begin with a letter (uppercase or lowercase) or an underscore (_).
2. The remaining characters can be letters, digits (0-9), or underscores.
3. Identifiers are case-sensitive.
4. An identifier cannot be a reserved keyword (like int, float, return, etc.).
5. Identifiers should not contain special characters (other than underscore) and cannot start with a digit.

❖ Implementation:

- **C Program to Check Validity of an Identifier:**

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


#define MAX_IDENTIFIER_LENGTH 100


// Function to check if a given identifier is valid

int is_valid_identifier(const char *identifier) {

    // Check if the identifier is empty

    if (identifier[0] == '\0') {

        return 0;  // Invalid

    }


    // Check the first character

    if (!isalpha(identifier[0]) && identifier[0] != '_')
{

        return 0;  // Invalid

    }


    // Check remaining characters

    for (int i = 1; i < strlen(identifier); i++) {
```

```
        if (!isalnum(identifier[i]) && identifier[i] !=
'_') {
            return 0;  // Invalid
        }
    }

    // List of reserved keywords in C
    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"
    };

    // Check against reserved keywords
    for (int i = 0; i < sizeof(keywords) /
sizeof(keywords[0]); i++) {
        if (strcmp(identifier, keywords[i]) == 0) {
            return 0; // Invalid
```

```
        }
    }
    return 1;  // Valid
}
int main() {
    char identifier[MAX_IDENTIFIER_LENGTH];
    printf("Enter an identifier: ");
    scanf("%s", identifier);
    if (is_valid_identifier(identifier)) {
        printf("'%s'    is    a    valid    identifier.\n",
identifier);
    } else {
        printf("'%s'  is  not  a  valid  identifier.\n",
identifier);
    }
    return 0;
}
```

- **Output:**

```
Enter an identifier: myVariable
'myVariable' is a valid identifier.
```

Practical – 4

Aim: Write a C program to simulate lexical analyzer for validating operators.

❖ **Description:**

- To create a C program that simulates a lexical analyzer for validating operators, we need to identify valid operators based on the rules of the C programming language. Common operators include:
- Arithmetic Operators: +, -, *, /, %
- Relational Operators: ==, !=, >, <, >=, <=
- Logical Operators: &&, ||, !
- Bitwise Operators: &, |, ^, ~, <<, >>
- Assignment Operators: =, +=, -=, *=, /=, %=
- Increment/Decrement Operators: ++, --

❖ **Implementation:**

- C Program to Simulate a Lexical Analyzer for Validating

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>



#define MAX_INPUT_LENGTH 256



// Function to check if the given string is a valid
operator
```

```
int is_operator(const char *token) {
    const char *operators[] = {
        "+", "-", "*", "/", "%", "==", "!=", ">", "<",
">=", "<=",
        "&&", "||", "!", "&", "|", "^", "~", "<<", ">>",
        "=", "+=", "-=", "*=", "/=", "%=", "++", "--"
    };

    for (int i = 0; i < sizeof(operators) /
sizeof(operators[0]); i++) {
        if (strcmp(token, operators[i]) == 0) {
            return 1;  // Valid operator
        }
    }
    return 0;  // Not a valid operator
}

int main() {
    char input[MAX_INPUT_LENGTH];
    printf("Enter an expression: ");
    fgets(input, sizeof(input), stdin); // Read a line
from standard input
```

```
    // Remove newline character from the end
    input[strcspn(input, "\n")] = '\0';


    char *token = strtok(input, " ");   // Tokenize the
input based on spaces
    printf("Identified operators:\n");
    // Loop through the tokens and check if they are
operators
    while (token != NULL) {
        if (is_operator(token)) {
            printf("%s\n", token);  // Print the valid
operator
        }
        token = strtok(NULL, " ");  // Get the next token
    }
    return 0;
}
```

- **Output:**

```
Enter an expression: a + b - c * d /
Identified operators:
+
-
*
/
```

Practical – 5

Aim: Study of Lexical Analyzer Generators: LEX and Flex.

❖ **Description:**

- A lexical analyzer (lexer) is a program that converts a sequence of characters (source code) into a sequence of tokens, which are meaningful sequences for the compiler or interpreter. Lexical analysis is the first phase of a compiler, and its primary purpose is to simplify the parsing phase by eliminating irrelevant information, such as whitespace and comments, and identifying the essential parts of the code (tokens).
- **LEX**
    - LEX is a tool for generating lexical analyzers. It was originally developed in the early 1970s and is primarily used in conjunction with Yacc (Yet Another Compiler Compiler), a parser generator.
    - LEX reads a specification file, which contains patterns (regular expressions) that describe the tokens to be recognized. It generates a C program that implements a finite state machine to recognize these tokens.
    - **Regular Expressions**: LEX uses regular expressions to define tokens.
    - **Rules Section**: The specification file contains a rules section where each rule associates a pattern with an action (code to execute when the pattern is matched).

- **Flexibility**: Supports character classes, actions for matched patterns, and multi-line comments.

- **Basic Structure of a LEX Specification**:

```
%{
/* C code declarations */
%}

%%
/* Rules and actions */
pattern1 { /* action for pattern1 */ }
pattern2 { /* action for pattern2 */ }

%%

int main(int argc, char **argv) {
    yylex(); // Calls the generated lexer
    return 0;
}
```

- **Example:**
  - A simple LEX specification that recognizes identifiers and numbers:

```
%{
#include <stdio.h>
%}
```

```
%%

[a-zA-Z_][a-zA-Z0-9_]*        {  printf("Identifier:
%s\n", yytext); }

[0-9]+                       { printf("Number: %s\n",
yytext); }

[ \t\n]+                   { /* Ignore whitespace */ }

.                         { printf("Unknown character:
%s\n", yytext); }



%%



int main() {

    yylex(); // Start lexical analysis

    return 0;

}
```

- **Flex (Fast Lexical Analyzer Generator)**
  - Flex is an open-source alternative to LEX, designed to be more efficient and flexible. It has become the standard lexical analyzer generator due to its enhanced features and performance.
  - Functionality: Similar to LEX, Flex takes a specification file as input and generates C code for a lexer. It produces faster and more efficient code and supports additional features like defining multiple patterns for a single token and user-defined actions.

- Performance: Flex-generated lexers are typically faster than those generated by LEX.
- Extended Regular Expressions: Supports additional constructs like repetition, optionality, and grouping.
- Debugging Options: Flex provides debugging options to help track the lexer's operation.
- Reentrant Lexers: Flex supports the generation of reentrant lexers, which are useful for parsing nested structures.
- **Basic Structure of a Flex Specification**:

```
%{
/* C code declarations */
%}

%%
/* Rules and actions */
pattern1 { /* action for pattern1 */ }
pattern2 { /* action for pattern2 */ }

%%

int main(int argc, char **argv) {
    yylex(); // Calls the generated lexer
    return 0;
}
```

- **Example**: A simple Flex specification that recognizes identifiers and numbers:

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z_][a-zA-Z0-9_]*    { printf("Identifier: %s\n", yytext); }
[0-9]+                    { printf("Number: %s\n", yytext); }
[ \t\n]+                  { /* Ignore whitespace */ }
.                         { printf("Unknown character: %s\n", yytext); }

%%

int main() {
    yylex(); // Start lexical analysis
    return 0;
}
```

Practical – 6

Aim: Implement following programs using Lex. a. Create a Lexer to take input from text file and count no of characters, no. of lines & no. of words. b. Write a Lex program to count number of vowels and consonants in a given input string. c. Write a Lex program to print out all numbers from the given file. d. Write a Lex program to printout all HTML tags in file. e. Write a Lex program which adds line numbers to the given file and display the same onto the standard output.

❖ **Lexer to Count Characters, Lines, and Words:**

- Lex Specification: count.l

```
%{
#include <stdio.h>
int char_count = 0;
int word_count = 0;
int line_count = 0;
%}
%%
\n               { line_count++; }
[ \t]+          { /* Ignore whitespace */ }
[a-zA-Z]+       { word_count++; }
.               { char_count++; }
%%
```

```
int main() {
    yylex();
    printf("Characters: %d\n", char_count);
    printf("Words: %d\n", word_count);
    printf("Lines: %d\n", line_count);
    return 0;
}
```

- Compiling and running

```
flex count.l
gcc -o count lex.yy.c -lfl
./count < input.txt
```

❖ Count Number of Vowels and Consonants:

- Lex Specification: vowels_consonants.l

```
%{
#include <stdio.h>

int  vowel_count  =  0;
int consonant_count = 0;
%}

%%
[aeiouAEIOU]    { vowel_count++; }
[a-zA-Z]       { consonant_count++; }
[ \t\n]+       { /* Ignore whitespace */ }

%%

int main() {
    yylex();
    printf("Vowels: %d\n", vowel_count);
    printf("Consonants: %d\n", consonant_count);
    return 0;
}
```

- Compiling and running

```
flex vowels_consonants.l
gcc -o vowels_consonants lex.yy.c -lfl
./vowels_consonants < input.txt
```

❖ Print Out All Numbers from the Given File:

- Lex Specification: numbers.l

```
%{
#include <stdio.h>
%}

%%
[0-9]+          { printf("Number: %s\n", yytext); }
[ \t\n]+        { /* Ignore whitespace */ }
.               { /* Ignore other characters */ }

%%

int main() {
    yylex();
    return 0;
}
```

- Compiling and running

```
flex numbers.l
gcc -o numbers lex.yy.c -lfl
./numbers < input.txt
```

❖ Print Out All HTML Tags in the File:

- Lex Specification: html_tags.l

```
%{
#include <stdio.h>
%}


%%
<[^>]+>         { printf("HTML Tag: %s\n", yytext); }
[ \t\n]+        { /* Ignore whitespace */ }
.               { /* Ignore other characters */ }


%%


int main() {
    yylex();
    return 0;
}
```

- Compiling and running

```
flex html_tags.l
gcc -o html_tags lex.yy.c -lfl
./html_tags < input.html
```

❖ Add Line Numbers to the Given File:

- Lex Specification: line_numbers.l

```
%{
#include <stdio.h>

int line_number = 1;
%}

%%
\n              { line_number++; printf("%d: ",
line_number); }
.               { putchar(yytext[0]); }

%%

int main() {
    printf("%d: ", line_number); // Print first line
number
    yylex();
    return 0;
}
```

- Compiling and running

```
flex line_numbers.l
gcc -o line_numbers lex.yy.c -lfl
./line_numbers < input.txt
```

Practical – 7

Aim: Write a program for constructing of LL (1) parsing.

❖ **Description:**

- Creating an LL(1) parser involves several steps, including defining a grammar, constructing the LL(1) parsing table, and implementing the parser itself. Below is an example that demonstrates how to construct an LL(1) parser in C, using a simple grammar.
- Steps to Implement LL(1) Parsing:
  - ➢ **Define the grammar.**
  - ➢ **Construct the first and follow sets.**
  - ➢ **Build the LL(1) parsing table.**
  - ➢ **Implement the parser.**

❖ **Implementation:**

- **ll_parser.c:**

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>



#define MAX 10

#define STACK_SIZE 100



char grammar[5][MAX] = {
```

```
    "E->TX",

    "X->+TX",

    "X->ε",

    "T->FY",

    "Y->*FY"

};


char terminals[MAX] = {'+', '*', '(', ')', 'i', '$'};

char non_terminals[MAX] = {'E', 'X', 'T', 'Y', 'F'};


// LL(1) Parsing table

char parse_table[5][6] = {

    {'E', 'T', 'X', ' ', ' ', ' '},

    {'X', '+', 'T', 'X', ' ', ' '},

    {'X', 'ε', ' ', ' ', ' ', ' '},

    {'T', 'F', 'Y', ' ', ' ', ' '},

    {'Y', '*', 'F', 'Y', ' ', ' '}

};


// Stack for the parser

char stack[STACK_SIZE];

int top = -1;
```

```
// Function prototypes
void push(char c);
char pop();
void print_stack();
void parse(char *input);
void  create_parse_table();

int main() {
    char input[MAX];

    // Input string
    printf("Enter the input string (use 'i' for
identifiers and '$' for end of input): ");
    scanf("%s", input);
    strcat(input, "$"); // Append end marker

    // Initialize stack with starting symbol
    push('$');
    push('E');

    // Parse the input string
    parse(input);
```

```
    return 0;
}

// Push function for stack
void push(char c) {
    if (top < STACK_SIZE - 1) {
        stack[++top] = c;
    } else {
        printf("Stack Overflow\n");
    }
}

// Pop function for stack
char pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
        printf("Stack Underflow\n");
        return '\0';
    }
}
```

```
// Print the current stack
void print_stack() {
    printf("Stack: ");
    for (int i = top; i >= 0; i--) {
        printf("%c ", stack[i]);
    }
    printf("\n");
}

// Parse function
void parse(char *input) {
    int i = 0; // Input index
    char symbol;

    while (1) {
        symbol = pop();
        print_stack();
        // If symbol is terminal
        if (symbol == input[i]) {
            printf("Matched: %c\n", symbol);
            i++;
```

```
            if (symbol == '$') break; // End of input
        } else if (symbol >= 'A' && symbol <= 'Z') {
// Non-terminal
            int row = symbol - 'A';
            int col = -1;

            // Find the column in the parse table
            for (int j = 0; j < sizeof(terminals); j++)
{
                if (terminals[j] == input[i]) {
                    col = j;
                    break;
                }
            }

            if (col == -1) {
                printf("Error: No matching rule for
%c\n", symbol);
                exit(1);
            }

            char production = parse_table[row][col];
            if (production != ' ') {
```

```
                if (production != 'ε') {
                    for (int j = strlen(production) -
1; j >= 0; j--) {
                        push(production[j]);
                    }
                }
            } else {
                printf("Error: No production found for
%c\n", symbol);
                exit(1);
            }
        } else {
            printf("Error: Unexpected symbol %c\n",
symbol);
            exit(1);
        }
    }

    printf("Input string is successfully parsed.\n");
}
```

- **Output:**

```
i+i*i$
Enter the input string (use 'i' for
identifiers and '$' for end of
input): i+i*i$
Stack: E $
Matched: i
Stack: X $
Stack: X $
Matched: +
Stack: T $
Matched: i
Stack: Y $
Stack: Y $
Matched: *
Stack: F $
Matched: i
Stack: $
Input string is successfully parsed.
```

Practical – 8

Aim: Implementation of Recursive Descent Parser without backtracking

Input: The string to be parsed.

Output: Whether string parsed successfully or not.

Explanation: You have to implement the recursive procedure for RDP for a typical grammar. The

production no. are displayed as they are used to derive the string.

❖ Description:

- A recursive descent parser is a top-down parser that uses a set of recursive procedures to process the input string according to a defined grammar. Below is a complete implementation of a recursive descent parser in C that parses a simple grammar without backtracking.
- **Example:**

```
1. E -> T E'
2. E' -> + T E' | ε
3. T -> F T'
4. T' -> * F T' | ε
5. F -> ( E ) | id
```

❖ Implementation:

- **Implementation of Recursive Descent Parser.**

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX 100

char input[MAX];    // Input string

int pos = 0;        // Current position in input

void E();           // Forward declarations

void E_prime();

void T();

void T_prime();

void F();

void match(char expected) {

    if (input[pos] == expected) {

        printf("Matched: %c\n", expected);

        pos++;

    } else {

        printf("Error: Expected %c, but found %c\n",
expected, input[pos]);

        exit(1);

    }

}
```

```
void E() {
    printf("Production: E -> T E'\n");
    T();
    E_prime();
}
void E_prime() {
    if (input[pos] == '+') {
        printf("Production: E' -> + T E'\n");
        match('+');
        T();
        E_prime();
    } else {
        printf("Production: E' -> ε\n");
    }
}
void T() {
    printf("Production: T -> F T'\n");
    F();
    T_prime();
}
void T_prime() {
    if (input[pos] == '*') {
```

```
        printf("Production: T' -> * F T'\n");
        match('*');
        F();
        T_prime();
    } else {
        printf("Production: T' -> ε\n");
    }
}
void F() {
    if (input[pos] == '(') {
        printf("Production: F -> ( E )\n");
        match('(');
        E();
        match(')');
    } else if (isalnum(input[pos])) { // Checking for
identifiers (id)
        printf("Production: F -> id\n");
        match(input[pos]);
    } else {
        printf("Error:    Unexpected    symbol    %c\n",
input[pos]);
        exit(1);
    }
```

```
}

int main() {

    printf("Enter the input string (use 'id' for
identifiers, e.g., a, b, c): ");

    scanf("%s", input);

    strcat(input, "$"); // Append end marker

    input[strlen(input)] = '\0'; // Null-terminate


    printf("Parsing the input string...\n");

    E();


    // Check if parsing is complete

    if (input[pos] == '$') {

        printf("Input string parsed successfully.\n");

    } else {

        printf("Error: Remaining input after parsing:
%c\n", input[pos]);

    }


    return 0;

}
```

- **Output:**

```
Enter the input string (use 'id' for identifiers, e.g., a,
b, c): a+b*c
Parsing the input string...
Production: E -> T E'
Production: T -> F T'
Production: F -> id
Matched: a
Production: E' -> + T E'
Matched: +
Production: T -> F T'
Production: F -> id
Matched: b
Production: T' -> * F T'
Matched: *
Production: F -> id
Matched: c
Production: T' -> ε
Production: E' -> ε
Input string parsed successfully.
```

Practical – 9

Aim: Write a program to implement LALR parsing.

❖ **Description:**

- Implementing an LALR (Look-Ahead LR) parser involves a more complex structure compared to simpler parsing techniques like recursive descent parsers. LALR parsers use a finite state machine to manage the parsing process, which requires constructing a parsing table based on a given grammar.
    - **Overview of LALR Parsing**:
        - ➢ **Grammar**: You need a context-free grammar (CFG) to define the language.
        - ➢ **LR(0) Items**: An LR(0) item is a production rule with a dot indicating how much of the rule has been seen.
        - ➢ **States**: The parser builds states based on the items.
        - ➢ **Parsing Table**: The table consists of action and goto functions, which guide the parser's decisions.

❖ **Implementing an LALR Parser in C:**

- Here's a basic implementation of an LALR parser in C. This program uses a predefined parsing table for the grammar and processes the input tokens.:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX_STACK_SIZE 100
```

```c
#define MAX_INPUT_SIZE 100


// Tokens
typedef enum {
    ID, PLUS, STAR, LPAREN, RPAREN, END, ERROR
} TokenType;


// Structure for parsing table
typedef struct {
    int action[10][10]; // Action table
    int gotoTable[10][10]; // Goto table
} ParsingTable;


// Global variables
char *input;
TokenType currentToken;
int position = 0;
int stack[MAX_STACK_SIZE];
int top = -1;


// Function prototypes
void error();
```

```
void advance();

void parse();

void push(int state);

int pop();

TokenType nextToken();

void printStack();


// LALR Parsing Table

ParsingTable table = {

    .action = {

        // State: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

        {2, ERROR, ERROR, 1, ERROR, ERROR},  //
State 0

        {ERROR, 3, ERROR, ERROR, ERROR, ERROR},
// State 1

        {ERROR, ERROR, 5, ERROR, ERROR, ERROR},
// State 2

        {ERROR, ERROR, ERROR, ERROR, 6, ERROR},
// State 3

        {ERROR, ERROR, ERROR, 4, ERROR, ERROR},
// State 4

        {7, ERROR, ERROR, ERROR, ERROR, ERROR},
// State 5
```

```
            {ERROR, 3, ERROR, ERROR, ERROR, ERROR},
        // State 6
            {ERROR, ERROR, 5, ERROR, ERROR, ERROR},
        // State 7
            {ERROR, ERROR, ERROR, 8, ERROR, ERROR},
        // State 8
            {ERROR, ERROR, ERROR, ERROR, ERROR, 9}
        // State 9
            },
            .gotoTable = {
         // State: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
                {0, 1, 2, -1, -1, -1}, // State 0
                {-1, -1, -1, -1, -1, -1}, // State 1
                {-1, -1, -1, -1, -1, -1}, // State 2
                {-1, -1, -1, -1, -1, -1}, // State 3
                {-1, -1, -1, -1, -1, -1}, // State 4
                {-1, -1, -1, -1, -1, -1}, // State 5
                {-1, -1, -1, -1, -1, -1}, // State 6
                {-1, -1, -1, -1, -1, -1}, // State 7
                {-1, -1, -1, -1, -1, -1}, // State 8
                {-1, -1, -1, -1, -1, -1}  // State 9
      }
  };
```

```
void error() {
    printf("Parsing error!\n");
    exit(1);
}

void advance() {
    currentToken = nextToken();
    if (currentToken == ERROR) {
        error();
    }
}

void parse() {
    push(0); // Initial state
    while (1) {
        int state = stack[top];
        if (currentToken == END && state == 9)
{
            printf("Input parsed
 successfully!\n");
           break;
        }
```

```
        int                 action              =
table.action[state][currentToken];
        if (action == ERROR) {
            error();
        } else if (action > 0) {
            printf("Shift   to   state:   %d\n",
action);
            push(action);
            advance();
        } else {
            printf("Reduce by production: E ->
E + T\n");
            top  -=  2;  //  Pop  2  symbols  from
stack
            state = stack[top]; // Current state
after popping
            int             gotoState           =
table.gotoTable[state][0]; // Goto E
            push(gotoState);
        }
    }
}

void push(int state) {
```

```
        if (top < MAX_STACK_SIZE - 1) {
            stack[++top] = state;
        } else {
            error();
        }
    }

    int pop() {
        if (top >= 0) {
            return stack[top--];
        } else {
            error();
        }
        return -1; // Should not reach here
    }

    TokenType nextToken() {
        while (input[position] == ' ') position++;
    // Skip spaces
        char ch = input[position];
        if (ch == '\0') return END;
        if (ch == '+') {
            position++;
```

```
            return PLUS;
        }
        if (ch == '*') {
            position++;
            return STAR;
        }
        if (ch == '(') {
            position++;
            return LPAREN;
        }
        if (ch == ')') {
            position++;
            return RPAREN;
        }
        if (isalnum(ch)) {
            position++;
            return ID; // Identifiers
        }
        return ERROR;
    }

    int main() {
```

```
    input = malloc(MAX_INPUT_SIZE);
    printf("Enter the input string (example: a + b * c): ");
    fgets(input, MAX_INPUT_SIZE, stdin);
    input[strcspn(input, "\n")] = 0; // Remove newline character

    advance(); // Start parsing
    parse(); // Parse input
    free(input);
    return 0;
}
```

- **Output:**

```
Enter the input string (example: a + b * c):
a + b * c
Shift to state: 1
Reduce by production: E -> E + T
Shift to state: 2
Shift to state: 3
Input parsed successfully!
```

Practical – 10

Aim: Write a program to implement operator precedence parsing.

❖ **Description:**

- Operator precedence parsing is a top-down parsing technique that uses a set of rules to decide which production to apply based on the precedence of operators in an expression. The technique involves parsing input expressions while considering operator precedence and associativity.
- **Overview of Operator Precedence Parsing:**
  - **Operator Precedence**:
    - Operators are assigned precedence levels, which determine the order of operations. For example, multiplication (*) has a higher precedence than addition (+).
  - **Associativity:**
    - This defines the order of operations for operators of the same precedence. For example, addition and subtraction are left associative.

❖ **Implementation:**

- Below is a simple C program that implements operator precedence parsing for the above grammar. The program takes an input expression and determines if it is valid based on the defined grammar.

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

#define MAX_INPUT_SIZE 100

char *input;      // Input expression
int position = 0; // Current position in input

// Function prototypes
void error();
void parseExpression();
void parseTerm();
void parseFactor();
void advance();
char currentChar();

// Function to handle errors
void error() {
    printf("Parsing error at position %d: unexpected
character '%c'\n", position, currentChar());
    exit(1);
}

// Function to get the next character
char currentChar() {
    return input[position];
```

```
}

// Function to advance the input position
void advance() {
    position++;
}

// Function to parse an expression (E)
void parseExpression() {
    parseTerm(); // Parse the first term

    while (currentChar() == '+' || currentChar() == '-
') {
        char op = currentChar(); // Get the operator
        advance(); // Advance past the operator
        parseTerm(); // Parse the next term
        printf("Parsed: %c\n", op); // Output the
operator
    }
}

// Function to parse a term (T)
void parseTerm() {
    parseFactor(); // Parse the first factor

    while (currentChar() == '*' || currentChar() ==
'/') {
        char op = currentChar(); // Get the operator
        advance(); // Advance past the operator
```

```
        parseFactor(); // Parse the next factor
        printf("Parsed: %c\n", op); // Output the
operator
    }
}

// Function to parse a factor (F)
void parseFactor() {
    if (isdigit(currentChar())) {
        printf("Parsed number: %c\n", currentChar());
        advance(); // Advance past the number
    } else if (currentChar() == '(') {
        advance(); // Advance past '('
        parseExpression(); // Parse the expression
inside parentheses
        if (currentChar() != ')') {
            error(); // Expecting ')'
        }
        advance(); // Advance past ')'
    } else {
        error(); // Unexpected character
    }
}

// Main function
int main() {
    input = malloc(MAX_INPUT_SIZE);
    printf("Enter an arithmetic expression: ");
    fgets(input, MAX_INPUT_SIZE, stdin);
```

```
    input[strcspn(input, "\n")] = 0; // Remove newline
character

    parseExpression(); // Start parsing
    printf("Input parsed successfully!\n");

    free(input);
    return 0;
}
```

- **Output:**

```
Enter an arithmetic expression: 3 + 5 * (2 - 8)
Parsed number: 3
Parsed number: 5
Parsed:  *
Parsed number: 2
Parsed number: 8
Parsed: -
Parsed: +
Input parsed successfully!
```

Practical – 11

Aim: To Study about Yet Another Compiler-Compiler (YACC) and Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / . –

❖ **Description:**

- YACC (Yet Another Compiler Compiler) is a tool used to generate parsers, which are used in compilers and interpreters. YACC takes a context-free grammar as input and produces source code in C for a parser that can recognize valid strings of that grammar. It is typically used in conjunction with Lex, a lexical analyzer, to create a complete compiler front-end.
- **Key Features of YACC:**
  - ➢ Grammar Specification: You define grammar rules in a high-level format.
  - ➢ Automatic Parser Generation: YACC generates C code for a parser based on the grammar.
  - ➢ Error Handling: YACC can handle errors gracefully during parsing.
  - ➢ Associativity and Precedence: You can specify operator precedence and associativity rules directly in the grammar.

❖ **Implementation:**

- Creating YACC and Lex Specifications for Arithmetic Expressions

- In this example, we will create a simple calculator that can recognize arithmetic expressions involving addition (+), subtraction (-), multiplication (*), and division (/).
- Step 1: Lex Specification (Lexer).
- Create a file named calc.l for the Lex specification.

```
%{
#include "y.tab.h"
%}


%%
[0-9]+            { yylval = atoi(yytext); return
NUMBER; }
[ \t\n]         { /* ignore whitespace */ }
"+"             { return '+'; }
"-"             { return '-'; }
"*"             { return '*'; }
"/"             { return '/'; }
"("             { return '('; }
")"             { return ')'; }
.               { return yytext[0]; } // return any
other character
%%
```

```
// Add main function to test lexer

int yywrap() {

    return 1;

}
```

- **Step 2: YACC Specification (Parser)**
- Create a file named calc.y for the YACC specification.

```
%{

#include <stdio.h>

#include <stdlib.h>



void yyerror(const char *s);

int yylex();

%}



%token NUMBER



%%

expr:   expr '+' term   { printf("%d\n", $1 + $3);
}

      | expr '-' term   { printf("%d\n", $1 - $3);
}

      | term              { $$ = $1; }

      ;
```

```
term:   term '*' factor { $$ = $1 * $3; }
      | term '/' factor { $$ = $1 / $3; }
      | factor           { $$ = $1; }
      ;

factor: NUMBER           { $$ = $1; }
      | '(' expr ')'     { $$ = $2; }
      ;

%%

// Error handling function
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

// Main function
int main() {
    printf("Enter arithmetic expressions (CTRL+D
to exit):\n");
    yyparse();
    return 0;
```

```
}
```

- **Output:**

```
3 + 4 * 2
(5 - 3) * 6
8 / 2 + 3
11
12
7
```

Practical – 12

Aim: Implement 3- address code(any one) w.r.t intermediate code for given infix expression.

- ❖ **Description:**
  - Three-address code is an intermediate representation used in compilers, where each instruction has at most three addresses or operands. It provides a simple way to express computations using variables, constants, and temporary values.
  - **Example of infix expression:**
    - ➢ a = b + c * d - e.
  - **Convert to Postfix:**
    - ➢ a = b c d * + e -
  - **Generate Three-Address Code:**
    - ➢ Now, we can generate the three-address code for the postfix expression. The code will include temporary variables to hold intermediate results.
    - ➢ Three-Address Code Generation Steps:
      - ➢ Identify operations in the postfix expression.
      - ➢ Use temporary variables to store intermediate results.
      - ➢ Create three-address code instructions for each operation.

❖ Example:

- Here's a simple C program that generates three-address code for the given expression:

```
t1 = c * d
t2 = b + t1
a = t2 - e
```

```c
#include <stdio.h>

int main() {
    // Given infix expression: a = b + c * d - e
    // Generate three-address code

    // Step 1: Generate code for c * d
    printf("t1 = c * d\n");

    // Step 2: Generate code for b + t1
    printf("t2 = b + t1\n");

    // Step 3: Generate code for t2 - e
    printf("a = t2 - e\n");

    return 0;
}
```