PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 8 (A)

Aim : To implement any parsing technique.

Objective: Develop a program to implement

a. Predictive parser
b. Operator precedence parser

Outcome: Students are able to understand various parsing techniques. Also they are able to implement a program to generate parsing table for respective technique.

Theory:

A special case of top-down parsing without backtracking is called a predictive parsing. While writing grammar if we eliminate left recursion from it, and left factoring the grammar, we can obtain a grammar that can be parsed by a recursive-descent parser without backtracking is called a predictive parser.

A non-recursive predictive parser is build using stack. The main problem in predictive parsing is, how to decide which production to be applied for a non-terminal. The non-recursive parser given in figure 8.12, looks for the production to be applied in parsing table which is constructed from certain grammars.

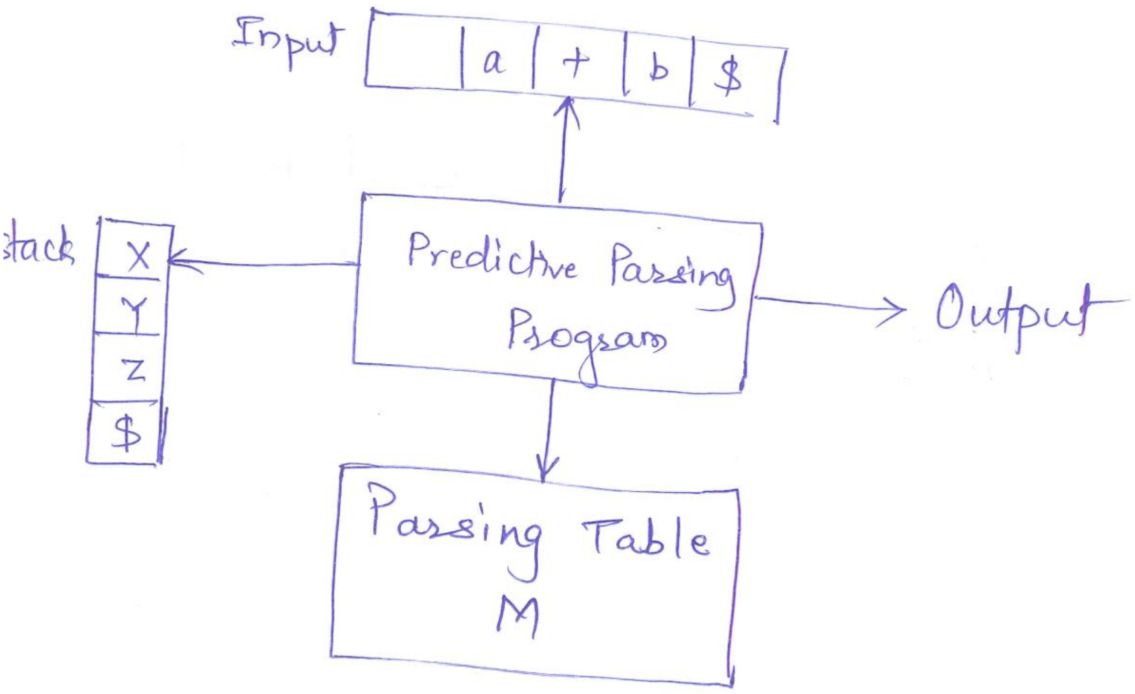


FIGURE: Model of a non-recursive predictive parser

Algorithm- Construction of predictive parsing table

Input- Grammar G

Output- Parsing Table

Method-

1. For each production A□α of the grammar do steps 2 and 3
2. For each terminal a in FIRST(α), add A□α to M[A,a]
3. If ε is in FIRST(α), add A□α M[A,b] for each terminal b in FOLLOW(A).
4. If ε is in FIRST(α) and $ is in FOLLOW(A), add A□α to M[A,$]
5. Make each undefined entry of M error.

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No. B44	Name: Intaza Chaudhary
Class:TE-B COMPS	Batch:B3
Date of Experiment:29/3/25	Date of Submission: 29/3/25
Grade:	

B.1 Software Code written by student:

Predictive parser:

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

// Grammar productions
char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };
char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };
char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };
char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };
char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };
char table[5][6][10];

// Function to map characters to table indices
int numr(char c) {
   switch (c) {
      case 'S': return 0;
      case 'A': return 1;
      case 'B': return 2;
      case 'C': return 3;
      case 'a': return 0;
      case 'b': return 1;
      case 'c': return 2;
      case 'd': return 3;
      case '$': return 4;
   }
   return 2;
}

int main() {
   int i, j, k;

   // Initialize parsing table with empty strings
   for (i = 0; i < 5; i++) {
      for (j = 0; j < 6; j++) {
         strcpy(table[i][j], " ");
      }
```

```
}

// Display the grammar used
printf("The following grammar is used for Parsing Table:\n");
for (i = 0; i < 7; i++) {
    printf("%s\n", prod[i]);
}

printf("\nConstructing Predictive Parsing Table...\n");

// Construct parsing table
for (i = 0; i < 7; i++) {
    k = strlen(first[i]);
    for (j = 0; j < k; j++) {
        if (first[i][j] != '@') {
            strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
        }
    }
}

for (i = 0; i < 7; i++) {
    if (strlen(pror[i]) == 1 && pror[i][0] == '@') {
        k = strlen(follow[i]);
        for (j = 0; j < k; j++) {
            strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
        }
    }
}

// Fill table headers
strcpy(table[0][0], " ");
strcpy(table[0][1], "a");
strcpy(table[0][2], "b");
strcpy(table[0][3], "c");
strcpy(table[0][4], "d");
strcpy(table[0][5], "$");
strcpy(table[1][0], "S");
strcpy(table[2][0], "A");
strcpy(table[3][0], "B");
strcpy(table[4][0], "C");

// Print the predictive parsing table
printf("\n--------------------------------------------------------\n");
for (i = 0; i < 5; i++) {
    for (j = 0; j < 6; j++) {
        printf("%-10s", table[i][j]);
        if (j == 5) {
            printf("\n--------------------------------------------------------\n");
        }
    }
    }

    return 0;
}
```

Operator precedence parser:

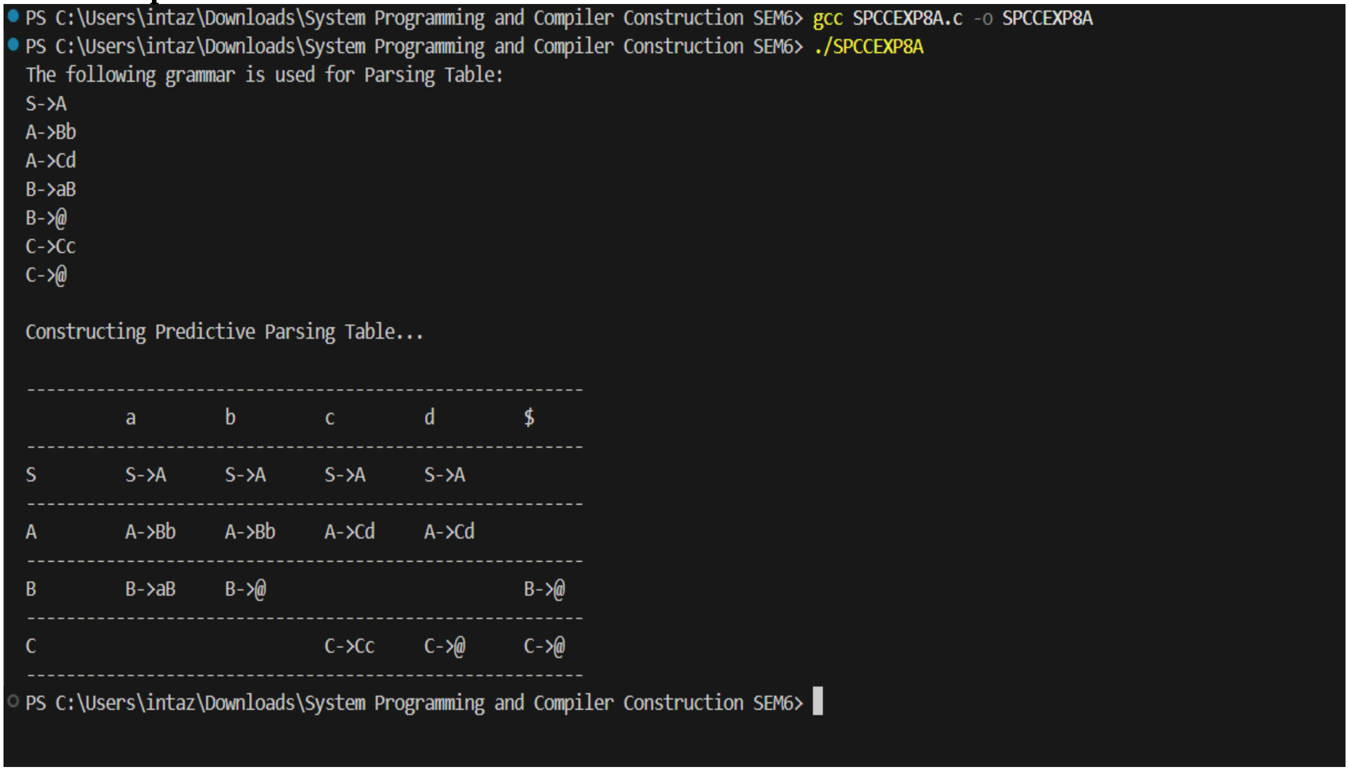
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Token types
typedef enum {
    END, ID, PLUS, MINUS, MULT, DIV, LPAREN, RPAREN
} TokenType;

// Token structure
typedef struct {
    TokenType type;
    char lexeme[100];
} Token;

// Global variables
Token current_token;
char *input;
int pos = 0;

// Updated precedence table with fixes for parentheses
// ' ' = error, '<' = shift, '>' = reduce, '=' = equal
char precedence_table[8][8] = {
    /*         END   ID    +     -     *     /     (     )    */
    /* END  */ {'=', '<', '<', '<', '<', '<', '<', ' '},
    /* ID   */ {'>', '>', '>', '>', '>', '>', ' ', '>'},
    /* +    */ {'>', '<', '>', '>', '<', '<', '<', '>'},
    /* -    */ {'>', '<', '>', '>', '<', '<', '<', '>'},
    /* *    */ {'>', '<', '>', '>', '>', '>', '<', '>'},
    /* /    */ {'>', '<', '>', '>', '>', '>', '<', '>'},
    /* (    */ {' ', '<', '<', '<', '<', '<', '<', '='},
    /* )    */ {'>', '>', '>', '>', '>', '>', ' ', '>'}
};

// Initialize the lexer
void init_lexer(char *source) {
    input = source;
    pos = 0;
}

// Get the next token
void get_next_token() {
    // Skip whitespace
```

```
while (input[pos] && isspace(input[pos])) pos++;

// Check for end of input
if (input[pos] == '\0') {
    current_token.type = END;
    strcpy(current_token.lexeme, "$");
    return;
}

// Check for identifier
if (isalnum(input[pos])) {
    int i = 0;
    while (isalnum(input[pos])) {
        current_token.lexeme[i++] = input[pos++];
    }
    current_token.lexeme[i] = '\0';
    current_token.type = ID;
    return;
}

// Check for operators
switch (input[pos]) {
    case '+':
        current_token.type = PLUS;
        strcpy(current_token.lexeme, "+");
        break;
    case '-':
        current_token.type = MINUS;
        strcpy(current_token.lexeme, "-");
        break;
    case '*':
        current_token.type = MULT;
        strcpy(current_token.lexeme, "*");
        break;
    case '/':
        current_token.type = DIV;
        strcpy(current_token.lexeme, "/");
        break;
    case '(':
        current_token.type = LPAREN;
        strcpy(current_token.lexeme, "(");
        break;
    case ')':
        current_token.type = RPAREN;
        strcpy(current_token.lexeme, ")");
        break;
    default:
        printf("Error: Unknown character %c\n", input[pos]);
        exit(1);
}
```

```
    pos++;
}

// Stack implementation for tokens
#define MAX_STACK 100
Token stack[MAX_STACK];
int top = -1;

void push(Token token) {
    if (top >= MAX_STACK - 1) {
        printf("Stack overflow\n");
        exit(1);
    }
    stack[++top] = token;
}

Token pop() {
    if (top < 0) {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack[top--];
}

Token peek() {
    if (top < 0) {
        printf("Stack is empty\n");
        exit(1);
    }
    return stack[top];
}

// Get the precedence relation between two tokens
char get_precedence(TokenType a, TokenType b) {
    return precedence_table[a][b];
}

// Special function to handle parenthesis reduction
// This function reduces all content between matching parentheses
// and the parentheses themselves
void reduce_parentheses() {
    // First, find the matching opening parenthesis
    int i = top;
    while (i >= 0 && stack[i].type != LPAREN) {
        i--;
    }

    if (i < 0) {
        printf("Error: Unmatched closing parenthesis\n");
        exit(1);
    }

    // Now reduce everything from opening parenthesis to top of stack
    top = i - 1;  // Remove the opening parenthesis and everything above it
}

// Helper function to print the current state with proper formatting
void print_current_state() {
    // Print stack
    printf("[ ");
    for (int i = 0; i <= top; i++) {
        printf("%s ", stack[i].lexeme);
    }
    printf("]");

    // Add padding to ensure columns align
    printf("%-30s", "");  // Clear previous text and add space

    // Print input with remaining tokens
    printf("[ %s", current_token.lexeme);
    if (input[pos] != '\0') {
        printf(" %s", input + pos);
    }
    printf(" ]");

    // Add padding before action
    printf("%-30s", "");
}

// Parse the expression
void parse() {
    Token dollar;
    dollar.type = END;
    strcpy(dollar.lexeme, "$");

    push(dollar); // Initial stack with $ at bottom
    get_next_token(); // Get the first token

    printf("Parsing steps:\n");
    printf("Stack%40sInput%40sAction\n", "", "");
    printf("--------------------------------------------------------------------------------\n");

    // Print initial state
    print_current_state();

    while (1) {
        // Handle special case for parentheses
        if (current_token.type == RPAREN && peek().type != LPAREN) {
            // Need to handle closing parenthesis specially
            printf("Handle closing paren\n");
            // Reduce until we find the matching opening parenthesis
            while (top > 0 && peek().type != LPAREN) {
                pop();
            }

            if (top <= 0 || peek().type != LPAREN) {
                printf("Error: Unmatched closing parenthesis\n");
                exit(1);
            }

            // Match the parentheses by popping the opening parenthesis
            pop();

            // Print new state
            print_current_state();

            // Continue with the next token
            get_next_token();
            continue;
        }

        // Get precedence relation
        char relation = get_precedence(peek().type, current_token.type);

        switch (relation) {
            case '<': // Shift
                printf("Shift\n");
                push(current_token);
                get_next_token();
                // Print new state
                print_current_state();
                break;

            case '=': // Equal (for parentheses or END matching)
                if (current_token.type == END && peek().type == END) {
                    printf("Accept\n");
                    printf("--------------------------------------------------------------------------------
\n");
                    printf("Parsing completed successfully\n");
                    return;
                } else if (peek().type == LPAREN && current_token.type == RPAREN) {
                    printf("Match parentheses\n");
                    // Pop the opening parenthesis
                    pop();
                    // Get the next token
                    get_next_token();
                    // Print new state
                    print_current_state();
                } else {
```

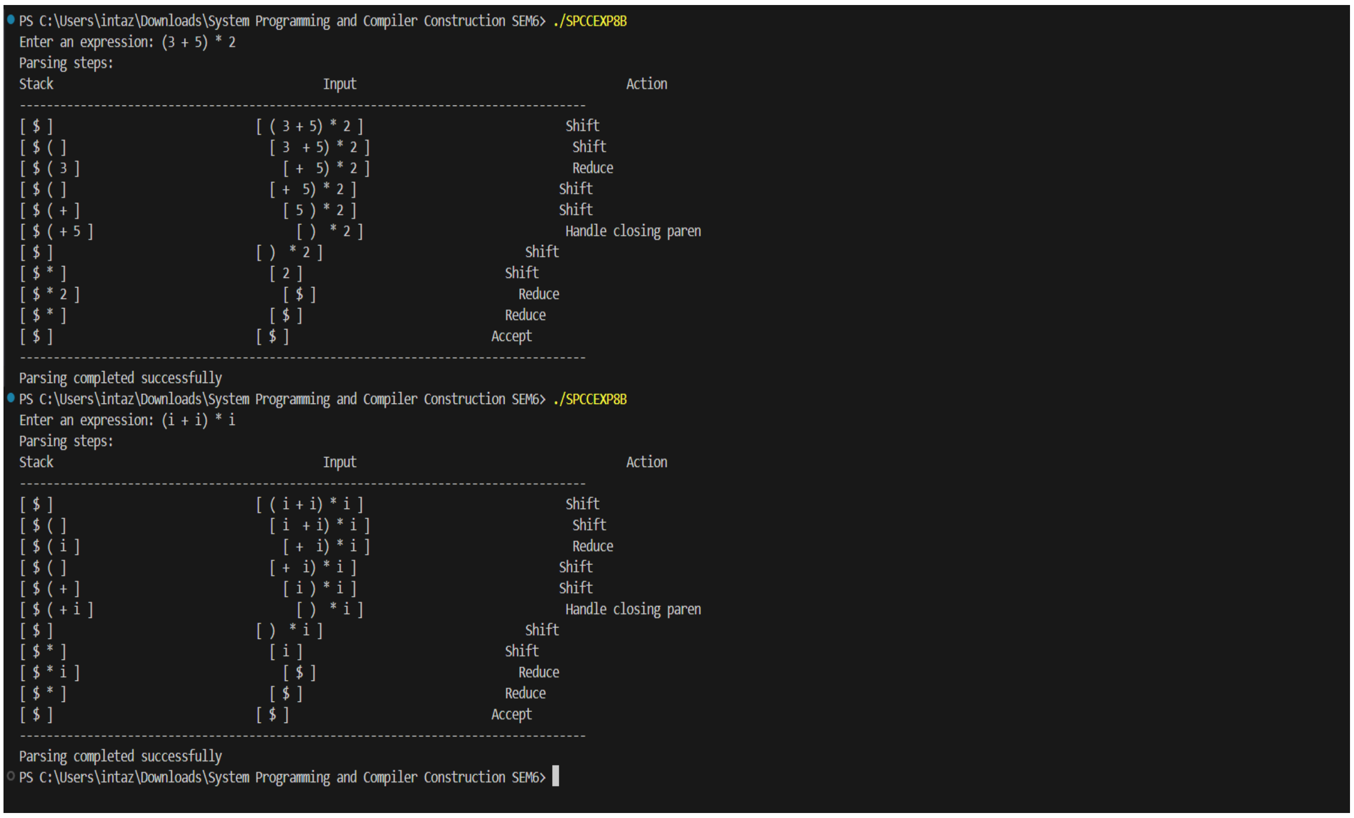
```
                push(current_token);
                get_next_token();
                // Print new state
                print_current_state();
            }
            break;

        case '>': { // Reduce
            // Print action before popping
            printf("Reduce\n");

            // In a full implementation, we'd use grammar rules
            // For this simple version, we just pop one token
            Token reduced = pop();

            // Print new state after reduction
            print_current_state();
            break;
        }

        case ' ': // Error
            // Special case for end of parenthesized expression
            if (peek().type == LPAREN && current_token.type == END) {
                printf("Reduce parenthesized expression\n");
                // Pop the opening parenthesis
                pop();
                // Print new state
                print_current_state();
            } else {
                printf("Error: No precedence relation between %s and %s\n",
                    peek().lexeme, current_token.lexeme);
                exit(1);
            }
            break;
        }
    }
}

int main() {
    char input_expression[1000];

    printf("Enter an expression: ");
    fgets(input_expression, sizeof(input_expression), stdin);

    // Remove newline character if present
    size_t len = strlen(input_expression);
    if (len > 0 && input_expression[len-1] == '\n') {
        input_expression[len-1] = '\0';
    }
    init_lexer(input_expression);
    parse();

    return 0;
}
```
B.2 Input and Output:

Predictive parser:



```
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> gcc SPCCEXP8A.c -o SPCCEXP8A
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> ./SPCCEXP8A
The following grammar is used for Parsing Table:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Constructing Predictive Parsing Table...

-------------------------------------------------------
          a         b         c         d         $
-------------------------------------------------------
S         S->A      S->A      S->A      S->A
-------------------------------------------------------
A         A->Bb     A->Bb     A->Cd     A->Cd
-------------------------------------------------------
B         B->aB     B->@                          B->@
-------------------------------------------------------
C                             C->Cc     C->@      C->@
-------------------------------------------------------
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6>
```

Operator precedence parser:



```
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> ./SPCCEXP8B
Enter an expression: (3 + 5) * 2
Parsing steps:
Stack                             Input                                       Action
--------------------------------------------------------------------------------
[ $ ]                             [ ( 3 + 5) * 2 ]                            Shift
[ $ ( ]                             [ 3  + 5) * 2 ]                            Shift
[ $ ( 3 ]                             [ +  5) * 2 ]                            Reduce
[ $ ( ]                             [ +  5) * 2 ]                            Shift
[ $ ( + ]                             [ 5 ) * 2 ]                            Shift
[ $ ( + 5 ]                             [ )  * 2 ]                            Handle closing paren
[ $ ]                             [ )  * 2 ]                            Shift
[ $ * ]                             [ 2 ]                            Shift
[ $ * 2 ]                             [ $ ]                            Reduce
[ $ * ]                             [ $ ]                            Reduce
[ $ ]                             [ $ ]                            Accept
--------------------------------------------------------------------------------
Parsing completed successfully
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> ./SPCCEXP8B
Enter an expression: (i + i) * i
Parsing steps:
Stack                             Input                                       Action
--------------------------------------------------------------------------------
[ $ ]                             [ ( i + i) * i ]                            Shift
[ $ ( ]                             [ i  + i) * i ]                            Shift
[ $ ( i ]                             [ +  i) * i ]                            Reduce
[ $ ( ]                             [ +  i) * i ]                            Shift
[ $ ( + ]                             [ i ) * i ]                            Shift
[ $ ( + i ]                             [ )  * i ]                            Handle closing paren
[ $ ]                             [ )  * i ]                            Shift
[ $ * ]                             [ i ]                            Shift
[ $ * i ]                             [ $ ]                            Reduce
[ $ * ]                             [ $ ]                            Reduce
[ $ ]                             [ $ ]                            Accept
--------------------------------------------------------------------------------
Parsing completed successfully
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6>
```

B.3 Observations and learning:

During the experiment, we implemented predictive and operator precedence parsing techniques. Predictive parsing relies on constructing a parsing table to determine which production rule to apply, eliminating the need for backtracking. It was observed that the correctness of the parsing table depends on eliminating left recursion and left factoring from the given grammar.

For operator precedence parsing, we noted that it uses precedence relations between operators to decide the parsing steps. The experiment helped in understanding the importance of FIRST and FOLLOW sets in constructing predictive parsers. Furthermore, debugging errors in the parsing table was crucial in ensuring the correct functioning of the parser

B.4 Conclusion:

The experiment demonstrated the working principles of predictive and operator precedence parsers. Predictive parsing, a type of top-down parsing, uses a parsing table to determine the next production rule, eliminating backtracking. This approach is efficient when the grammar is LL(1), meaning it can be parsed using only one lookahead symbol.

B.5 Question of Curiosity

1) What is the mechanism of Top-Down parser ?
A top-down parser starts from the start symbol of the grammar and attempts to derive the input string by applying production rules. It constructs the parse tree from the root to the leaves. The key mechanisms include:

Recursive Descent Parsing: Uses a set of recursive functions to process input. It may involve backtracking unless the grammar is LL(1).

Predictive Parsing: A special form of recursive descent parsing that eliminates backtracking by using a parsing table. It works efficiently for LL(1) grammars.

2) How do you recognize LL(1) grammar ?

A grammar is LL(1) if it can be parsed using a single lookahead symbol and does not require backtracking. It satisfies these conditions:

The FIRST sets of different productions for a non-terminal must not overlap.

If a production derives ε (epsilon), the FIRST set of that production must not overlap with the FOLLOW set of the corresponding non-terminal.

The grammar should not have left recursion or common prefixes (left factoring is required).

3) What are the key differences between recursive and non recursive-descent parsers ?

Feature	Recursive Descent Parser	Non-Recursive Descent Parser
Method	Uses recursive functions to parse input	Uses a stack and parsing table
Backtracking	May require backtracking unless optimized (LL(1) avoids it)	Does not require backtracking
Memory Usage	Can lead to deep recursion and high memory consumption	More memory efficient due to iterative approach
Parsing Table	Not required	Requires a parsing table for decision making
Ease of Implementation	Easier to implement manually for small grammars	More complex due to table construction
Grammar Requirement	Works for general grammars but may need modifications to avoid backtracking	Requires an LL(1) grammar (left recursion and ambiguity must be removed)
Performance	Slower if backtracking occurs	Faster and more predictable
Error Handling	Harder to handle errors efficiently	Provides better error detection using parsing table