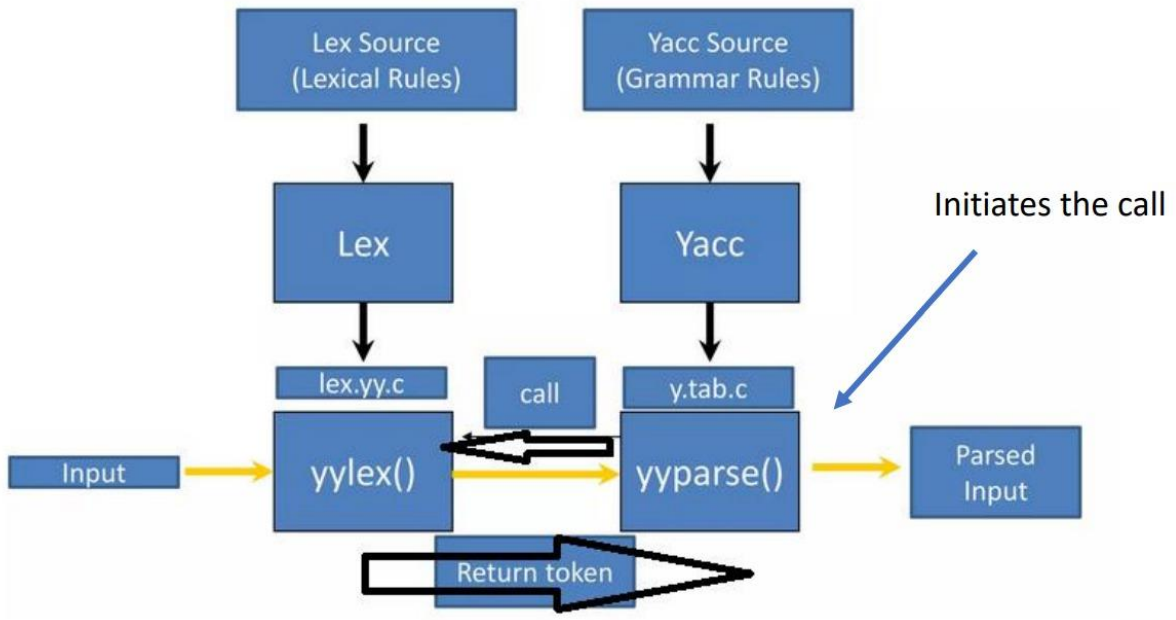
Name: Aravind Ashokan
Roll No: 16

DIAGRAM:



Date: 18.09.2025

EXPERIMENT- 2.5

LEX and YACC

AIM:

To understand LEX and YACC tools.

THEORY:

LEX is a computer program that generates lexical analysers ("scanner" or "lexers"). It is commonly used with the YACC parser generator and is the standard lexical analyser generator on many Unix and Unix-like systems. LEX reads an input stream specifying the lexical analyser and writes source code which implements the lexical analyser in the C programming language.
Structure of Lex Programs:A LEX program is organized into three main sections: Declarations, Rules, and Auxiliary Functions. Each section serves a distinct purpose in defining the behaviour of the lexical analyser.
1.Declarations: The Declarations section comprises two parts:

- Regular definitions: Define macros and shorthand notations for regular expressions to simplify rule definitions.
- Auxiliary Declarations: Contains C code such as header file inclusions, function prototypes, and global variable declarations. This code is enclosed within %{ and %} and is copied verbatim into the generated lex.yy.c file.

2.Rules: The Rules section consists of pattern-action pairs, where each pair defines a regular expression pattern to match and the corresponding C code to execute upon a successful match.

3.Auxiliary Functions: The Auxiliary Functions section includes additional C code that is not part of the rules. This typically contains the main function and any helper functions required by the lexer. This code is placed after the second %% and is directly copied into the lex.yy.c file.

yylex(): The 'yylex()' function, defined by Lex in the 'lex.yy.c' file, reads the input stream, matches it against regular expressions, and executes the corresponding actions for each match. It also generates tokens for further processing by parsers or other components, though the programmer must explicitly invoke 'yylex()' within the auxiliary functions of the LEX program.

yywrap(): The function yywrap() is called by yylex() when the end of an input file is reached. If yywrap() returns 0, scanning continues; if it returns a non-zero value, yylex() terminates and returns 0.

COMPILATION STEPS:

Step-1: Start
Step-2: Create a file with a .l and write your LEX.
Step-3: Give the command 'lex simple_calc.l'. This command generates a C source file named lex.yy.c.
Step-4: 'gcc -o simple_calc lex.yy.c -lfl' to compile the generated C code.
Step-5: Running the Executable Execute the compiled program using: './simple_calc'

YACC (Yet Another Compiler Compiler) generates LALR parsers from formal grammar specifications. It is often used with Lex for building compilers and interpreters, handling the syntactic analysis phase. YACC reads a grammar file and produces a C source parser that processes token sequences (typically from LEX) to check if they follow the grammar. The modern counterpart of YACC is Bison.

COMPILATION STEPS:

Step 1: Create a file with a .y and write your YACC.
Step 2: Give the command 'yacc -d simple_calc.y'. This command generates a C source file named y.tab.c.
Step 3: Use 'gcc -o simple_calc lex.yy.c -lfl' to compile the generated C code.
Step 4: Running the Executable Execute the compiled program using: './simple_calc'

Structure of YACC Program

%{
C Declarations
%}
Yacc Declarations
%%
Grammar Rules
%%
Additional Code (Comments enclosed in /*....*/ may appear in any section)

RESULT:

Familiarized ourselves with the working of YACC and LEX tools for compiler design.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

grammar.c:

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

struct ProductionRule{
    char symbol;
    char expression[20];
};

struct Grammar{
    char startState;
    char* non_terminals;
    char* terminals;
    struct ProductionRule* rules;
    int production_num;
};

struct LMDStackNode {
    struct ProductionRule rule;
    struct LMDStackNode* next;
};

struct LMDStackNode* head = NULL;

void free_grammar(struct Grammar* g){
    if (!g) return;
    if (g->non_terminals) free(g->non_terminals);
    if (g->terminals) free(g->terminals);
    if (g->rules) free(g->rules);
    free(g);
}
int find_index(char s[], char c){
    int n = strlen(s);
    for (int i=0;i<n;++i){
        if (s[i]==c){
            return i;
```

Date: 18.09.25

EXPERIMENT 3.1
OPERATOR PRECEDENCE PARSER

AIM:

Develop an operator precedence parser for a given language.

ALGORITHM:

Step 0: Start

Step 1: Input Grammar

- **1.1:** Input the number of non-terminals, terminals, and production rules.
- **1.2:** Input the start symbol of the grammar.
- **1.3:** Input the list of non-terminals.
- **1.4:** Input the list of terminals.
- **1.5:** Input each production rule in the form A->xyz.
- **1.6:** Store the grammar in appropriate data structures.

Step 2: Input Operator Precedence Table:

- **2.1:** Construct a table of size (no_of_terminals + 1) × (no_of_terminals + 1)
- **2.2:** Fill the table entries with precedence relations (<, >, =, A)

Step 3: Initialize Stacks

- **3.1:** Create two stacks:
  - **3.1.1:** Input Stack that contains the input string followed by end marker $.
  - **3.1.2:** Output Stack that is initially empty.
- **3.2:** Push input string characters into Input Stack (right to left).

Step 4: Parsing Loop (Repeat until accepted or rejected)

- **4.1:** Find Action
  - **4.1.1:** Let lhs = nearest terminal on top of Output Stack.
  - **4.1.2 :** Let rhs = nearest terminal on top of Input Stack.
  - **4.1.3:** Look up precedence table entry [lhs, rhs] to determine action.
- **4.2:** Perform Action

```
        }
    }
    return -1;
}
bool str_contains(char str[],char c){
    return find_index(str,c)!=-1;
}
void add_str(char str[], char c){
    int n = strlen(str);
    for (int i=0;i<n;++i){
        if (str[i]==c){
            return ;
        }
    }
    str[n] = c;
    str[n+1] = '\0';
}
bool validTerminal(struct Grammar* g, char c){
    return str_contains(g->terminals,c);
}

bool validNonTerminal(struct Grammar* g, char c){
    return str_contains(g->non_terminals,c);
}

bool validInput(struct Grammar* g, char input[]){
    int n = strlen(input);
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i])){
            return true;
        }
    }
```

4.2.1: If action = < (Shift)

4.2.1.1: Pop top symbol from Input Stack.

4.2.1.2: Push it into Output Stack.

4.2.1.3: Print current stack states.

4.2.2: If action = > or = (Reduce)

4.2.2.1: Check if top of Output Stack matches RHS of any production.

4.2.2.2: If matched then pop RHS symbols, push LHS non-terminal onto Output Stack, record applied production and print current stack states.

4.2.2.3: If no match then reject.

4.2.3: If action = A (Accept)

4.2.3.1: Check if Input Stack is empty and Output Stack contains only start symbol.

4.2.3.2: If true then print “String Accepted”.

4.2.3.3: Else print “String Rejected”.

Step 5: Output Derivation

5.1: Print the sequence of productions used during reductions.

5.2: This represents the Rightmost Derivation (RMD) of the input string.

Step 6: Stop

```
    return true;
}

bool validExpansion(struct Grammar* g, char input[]){
    int n = strlen(input);
    if (n==1 && input[0]=='e') return true;
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i]) && !validNonTerminal(g,input[i])){
            return true;
        }
    }
    return true;
}
struct Grammar* read_grammar() {
    int num_non_terminal, num_terminal, num_production_rule;
    scanf("%d %d %d",&num_non_terminal,&num_terminal,&num_production_rule);
    struct Grammar* g = malloc(sizeof(struct Grammar));
    if (!g){
        printf("Coulnd't create grammar\n");
        return NULL;
    }
    scanf(" %c",&g->startState);
    if (g->startState==EOF){
        printf("Reached EOF when reading start state\n");
        free_grammar(g);
        return NULL;
    }
    g->production_num = num_production_rule;
    //Read non terminals
    g->non_terminals = malloc(sizeof(char)*num_non_terminal);
    if (!g->non_terminals){
```

```
        printf("Couldnt' allocate non terminals\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_non_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading non terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->non_terminals[i] = c;
    }
    g->non_terminals[num_non_terminal] = '\0';
    //Read terminals
    g->terminals = malloc(sizeof(char)*num_terminal);
    for (int i=0;i<num_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->terminals[i] = c;
    }
    g->terminals[num_terminal] = '\0';

    //Read Production Rules
}
```

```
    g->rules = malloc(sizeof(struct ProductionRule)*num_production_rule);
    if (!g){
        printf("Error reading production rules\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_production_rule;++i){
        char rule[20];
        scanf("%s",rule);
        sscanf(rule,"%c->%s",&(g->rules[i].symbol),&g->rules[i].expression);
        if (!validNonTerminal(g,g->rules[i].symbol) || !validExpansion(g,g-
>rules[i].expression)){
            printf("Production rule %s invalid\n",rule);
            if (!validNonTerminal(g,g->rules[i].symbol)){
                printf("Invalid symbol on LHS\n");
            }
            if (!validExpansion(g,g->rules[i].expression)){
                printf("Invalid expression on RHS");
            }
            free_grammar(g);
            return NULL;
        }}
    return g;
}
void push_derivation(struct ProductionRule r){
    struct LMDStackNode* n = malloc(sizeof(struct LMDStackNode));
    n->next = head;
    n->rule = r;
    head = n;
}
bool empty_derivation(){
```

```
if (head) return false;
    return true;
}
void pop_derivation(){
    if (!head) return;
    struct LMDStackNode* n = head->next;
    free(head);
    head = n;
}
struct ProductionRule top_derivation(){
    return head->rule;
}
void print_delete_derivation(){
    if (empty_derivation()) return;
    struct ProductionRule p = top_derivation();
    pop_derivation();
    print_delete_derivation();
    printf("%c->%s\n",p.symbol,p.expression);
    printf("\n");
        }
```

stack.c:

```
#include "grammar.c"
struct StackNode{
    char symbol;
    char firstTerminal;
    struct StackNode* next;
};


bool emptyStack(struct StackNode** indirect){
    if (*indirect) return false;
    return true;
}
```

```
void stackPush(struct StackNode** indirect,char symbol, bool terminal){
    char firstTerminal = '$';
    if (terminal){
        firstTerminal = symbol;
    } else if (*indirect){
        firstTerminal = (*indirect)->firstTerminal;
    }
    struct StackNode* st = malloc(sizeof(struct StackNode));
    st->symbol = symbol;
    st->firstTerminal = firstTerminal;
    st->next = *indirect;
    *indirect = st;
}

void popStack(struct StackNode** indirect){
    if (*indirect){
        struct StackNode* st = *indirect;
        *indirect = st->next;
        free(st);
    }
}

char stackTopValue(struct StackNode** indirect){
    if (*indirect){
        return (*indirect)->symbol;
    }
    return '$';
}
char stackTerminal(struct StackNode** indirect){
    if (!emptyStack(indirect)){
        return (*indirect)->firstTerminal;
    }
    return '$';
}
void freeStack(struct StackNode** indirect){
    while (!emptyStack(indirect)){
        popStack(indirect);
    }
}

void printState(struct StackNode** indirect){
```

```
while (*indirect){
        printf("%c",(*indirect)->symbol);
        // printf("(%c,%c)",(*indirect)->symbol,(*indirect)->firstTerminal);
        indirect = &((*indirect)->next);
    }
    printf("$");
}
```

shift_reduce_common.c:

```
#include "stack.c"
bool match(struct StackNode** indirect, char s[]){
    int n = strlen(s);
    struct StackNode* current = *indirect;

    for (int i=n-1;i>=0;--i){
        if (!current){
            return false;
        }
        char lhs = current->symbol;
        char rhs = s[i];
        if (lhs!=rhs){
            return false;
        }

        current = current->next;
    }

    // Don't remove matched nodes here, that happens in reduce()

    return true;
}

bool shift(struct StackNode** inputStack,struct StackNode** outputStack){
    if (!emptyStack(inputStack)){
        stackPush(outputStack,stackTopValue(inputStack),true);
        popStack(inputStack);
        printf("Action: Shift Input: ");
        printState(inputStack);
        printf(" Output: ");
        printState(outputStack);
        printf("\n");
        return true;
    }
```

```
    return false;
}

bool reduce(struct StackNode** outputStack,struct StackNode** inputStack,struct
Grammar* g){
    int np = g->production_num;
    bool res = false;
    for (int p=0;p<np;++p){
        char* expression = g->rules[p].expression;
        char symbol = g->rules[p].symbol;
        if (match(outputStack,expression)){
            int n = strlen(expression);
            while (n-->0){
                popStack(outputStack);
            }
            stackPush(outputStack,symbol,false);
            push_derivation(g->rules[p]);
            res = true;
            printf("Action: Reduce Input: ");
            printState(inputStack);
            printf(" Output: ");
            printState(outputStack);
            printf("\n");
        }
    }
    return res;
}
void derivation_parse(){
    printf("The RMD is as follows:\n");
    while (!empty_derivation()){
        struct ProductionRule r = top_derivation();
        printf("%c->%s\n",r.symbol,r.expression);
        pop_derivation();
    }
}
```

operator.c:

```
#include "shift_reduce_common.c"

char ** read_precedence(struct Grammar* g){
    int n = strlen(g->terminals)+1;
    char ** table = malloc(sizeof(char*)*n);
    for (int i=0;i<n;++i){
```

```
        table[i] = malloc(sizeof(char)*n);
        scanf("%s",table[i]);
    }
    return table;
}
void free_precedence(char** table, int n){
    for (int i=0;i<n;++i){
        free(table[i]);
    }
    free(table);
}
char findAction(struct Grammar* g,char**table, struct StackNode** outputStack,struct
StackNode** inputStack){
    char lhs = stackTerminal(outputStack);
    char rhs = stackTerminal(inputStack);
    int l = find_index(g->terminals,lhs);
    int r = find_index(g->terminals,rhs);
    if (l<0) l = strlen(g->terminals);
    if (r<0) r = strlen(g->terminals);
    return table[l][r];
}
void parse(struct Grammar* g, char** table,char input[20]){
    struct StackNode * inputHead = NULL;
    struct StackNode * stackHead = NULL;

    struct StackNode** inputStack = &inputHead;
    struct StackNode** outputStack = &stackHead;
    int n = strlen(input);
    for (int i=n-1;i>=0;--i){
        stackPush(inputStack,input[i],true);
    }
    char action = '=';
    int max_iterations = 1000;
    while (max_iterations-->0 && action!='A'){
        action = findAction(g,table,outputStack,inputStack);
        switch(action){
            case '>':
            case '=':
                //Same for left associative grammar
                if (!reduce(outputStack,inputStack,g)){
                    max_iterations = 0;
                }
                break;
```

```
            case '<':
                if (!shift(inputStack,outputStack)){
                    max_iterations = 0;
                }
                break;
             case 'A': //Accept
                if (emptyStack(inputStack) && !emptyStack(outputStack)
                && !stackHead->next && stackTopValue(outputStack)==g->startState){
                    //Valid input
                    printf("String Accepted\n");
                } else {
                    printf("String rejected\n");
                }
                break;
        }
    }
    derivation_parse();
    freeStack(inputStack);
    freeStack(outputStack);
}
int main(){
    struct Grammar* g = read_grammar();
    char ** table = read_precedence(g);
    int n = strlen(g->terminals)+1;
    char input[20];
    scanf("%19s",input);
    if (validInput(g,input)){
        parse(g,table,input);
    } else {
        printf("Invalid input\n");
    }
 free_precedence(table,n);
    free_grammar(g);
    return 0;
}
```

input.txt:

```
1 3 3
E
E
i+*
E->E+E
```

E->E*E
E->i
=>>>
<><>
<><>
<<<A
i+i*i

OUTPUT:

Action: Shift Input: +i*i$ Output: i$
Action: Reduce Input: +i*i$ Output: E$
Action: Shift Input: i*i$ Output: +E$
Action: Shift Input: *i$ Output: i+E$
Action: Reduce Input: *i$ Output: E+E$
Action: Shift Input: i$ Output: *E+E$
Action: Shift Input: $ Output: i*E+E$
Action: Reduce Input: $ Output: E*E+E$
Action: Reduce Input: $ Output: E+E$
Action: Reduce Input: $ Output: E$
String Accepted
The RMD is as follows:
E->E+E
E->E*E
E->i
E->i
E->i

RESULT:

Implemented the program for developing an operator precedence parser for a given language.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
int n;
char prods[50][50];
char firsts[26][50];
int is_first_done[26];
char follows[26][50];
int is_follow_done[26];

int isTerminal(char c)
{
    if (c < 65 || c > 90)
        return 1;
    return 0;
}
void first(char nonterm)
{
    int index = 0;
    char curr_firsts[50];
    for (int i = 0; i < n; i++)
    {
        if (prods[i][0] == nonterm)
        {
            int curr_prod_index = 2;
            int flag = 0;
            while (prods[i][curr_prod_index] != '\0' && flag == 0)
            {
                flag = 1;
                if (isTerminal(prods[i][curr_prod_index]))
                {
                    curr_firsts[index] = prods[i][2];
                    index++;
                    break;
                }
                if (!is_first_done[prods[i][curr_prod_index] - 65])
                    first(prods[i][curr_prod_index]);
                int in = 0;
```

Date: 10.09.25

EXPERIMENT 3.2
SIMULATION OF FIRST AND FOLLOW

AIM:

Write a program to simulate First and Follow of any given grammar.

ALGORITHM:

Step 0: Start

Step 1: Declare an integer variable n for number of productions

Step 2: Declare 2D character arrays, prods[][] to store productions, firsts[][] to store FIRST sets and follows[][] to store FOLLOW sets

Step 3: Declare integer arrays, is_first_done[26] to track completion status of FIRST sets and is_follow_done[26] to track completion status of FOLLOW sets

Step 4: Read and store value in n, the no of productions

Step 5: Read the productions and store it in prods

Step 6: For each index i = 0 to 25

6.1: Set is_first_done[i] = 0

6.2: Set is_follow_done[i] = 0

Step 7: Compute FIRST sets

7.1: For each production of the nonterminal:

7.1.1: Scan the symbols on the right-hand side from left to right.

7.2.2: If the symbol is a terminal, add it to the FIRST set and stop scanning this production.

7.2.3: If the symbol is a non-terminal, recursively add all symbols from its FIRST set except epsilon.

7.2.4: If the non-terminal's FIRST set contains epsilon, continue scanning the next symbol.

7.2.5: If all symbols can derive epsilon, include epsilon in the FIRST set.

Step 8: Compute FOLLOW sets

8.1: For each non-terminal, compute its FOLLOW set:

```
            while (firsts[prods[i][curr_prod_index] - 65][in] != '\0')
                {
                    curr_firsts[index] = firsts[prods[i][curr_prod_index] - 65][in];
                    if (firsts[prods[i][curr_prod_index] - 65][in] == 'e')
                    {
                        curr_prod_index++;
                        flag = 0;
                    }
                    index++;
                    in++;
                }
            }
        }
    }
    curr_firsts[index] = '\0';
    index++;
    strcpy(firsts[nonterm - 65], curr_firsts);
    is_first_done[nonterm - 65] = 1;
}

void follow(char nonterm)
{
    int index = 0;
    char curr_follows[50];
    if (nonterm == prods[0][0])
    {
        curr_follows[index] = '$';
        index++;
    }
    for (int j = 0; j < n; j++)
    {
        int k = 2;
        int include_lhs_flag;
        while (prods[j][k] != '\0')
        {
            include_lhs_flag = 0;
            if (prods[j][k] == nonterm)
            {
                if (prods[j][k + 1] != '\0')
                {
                    if (isTerminal(prods[j][k + 1]))
                    {
                        curr_follows[index] = prods[j][k + 1];index++;
```

8.1.2: For each occurrence of the non-terminal in productions:

8.1.2.1: If it is followed by symbols, add the FIRST set of the next symbol (excluding epsilon).

8.1.2.2: If it is at the end or followed only by symbols that can derive epsilon, add the FOLLOW set of the left-hand side nonterminal.

8.1.2.3: Use recursion to compute FOLLOW sets as needed.

8.1.3: If the symbol is a non-terminal, recursively add all symbols from its FIRST set except epsilon.

8.1.4: If the non-terminal's FIRST set contains epsilon, continue scanning the next symbol.

8.1.5: If all symbols can derive epsilon, include epsilon in the FIRST set.

Step 9: Print the FIRST and FOLLOW sets for all non-terminals.

Step 10: Stop

```
            break;
        }
        int in = 0;
        while (firsts[prods[j][k + 1] - 65][in] != '\0')
        {
            if (firsts[prods[j][k + 1] - 65][in] == 'e')
            {
                include_lhs_flag = 1;
                in++;
                continue;
            }
            int temp_flag = 0;
            for (int z = 0; z < index; z++)
                if (firsts[prods[j][k + 1] - 65][in] == curr_follows[z])
                {
                    temp_flag = 1;
                    in++;
                    break;
                }
            if (temp_flag)
                continue;
            curr_follows[index] = firsts[prods[j][k + 1] - 65][in];
            index++;
            in++;
        }
    }
    if (prods[j][k + 1] == '\0' || include_lhs_flag == 1)
    {
        if (prods[j][0] != nonterm)
        {
            if (!is_follow_done[prods[j][0] - 65])
                follow(prods[j][0]);
            int x = 0;
            while (follows[prods[j][0] - 65][x] != '\0')
            {
                int temp_flag = 0;
                for (int z = 0; z < index; z++)
                    if (follows[prods[j][0] - 65][x] == curr_follows[z])
                    {
                        temp_flag = 1;
                        x++;
                        break;
                    }
                if (temp_flag)
```

```
                    continue;
                curr_follows[index] = follows[prods[j][0] - 65][x];
                index++;
                x++;
            }
        }
    }
    k++;
  }
 }
 curr_follows[index] = '\0';
 index++;
 strcpy(follows[nonterm - 65], curr_follows);
 is_follow_done[nonterm - 65] = 1;
}
int main()
{
 printf("Enter the number of productions: ");
 scanf("%d", &n);
 printf("Enter the productions: \n");
 for (int i = 0; i < n; i++)
  scanf("%s", prods[i]);
 for (int i = 0; i < 26; i++)
  is_first_done[i] = 0;
 for (int i = 0; i < n; i++)
  if (is_first_done[prods[i][0] - 65] == 0)
   first(prods[i][0]);
 for (int i = 0; i < n; i++)
  if (is_follow_done[prods[i][0] - 65] == 0)
   follow(prods[i][0]);
 printf("Firsts:\n");
 for (int i = 0; i < 26; i++)
 {
  if (is_first_done[i])
  {
   printf("%c : ", i + 65);
   for (int j = 0; firsts[i][j] != '\0'; j++)
   {
    printf("%c", firsts[i][j]);
    if (firsts[i][j + 1] != '\0')
     printf(", ");
   }
```

```
            printf("\n");
        }
    }

    printf("Follows:\n");
    for (int i = 0; i < 26; i++)
    {
        if (is_follow_done[i])
        {
            printf("%c : ", i + 65);
            for (int j = 0; follows[i][j] != '\0'; j++)
            {
                printf("%c", follows[i][j]);
                if (follows[i][j + 1] != '\0')
                    printf(", ");
            }
            printf("\n");
        }
    }}
```

OUTPUT:

NOTE: Use letter e as epsilon

```
Enter the number of productions: 8
Enter the productions:
E=TR
R=+TR
R=e
T=FY
Y=*FY
Y=e
F=(E)
F=i

Firsts:
E : (, i
F : (, i
R : +, e
T : (, i
Y : *, e
```

Follows:
E : $,)
F : *, +, $,)
R : $,)
T : +, $,)
Y : +, $,)

RESULT:

Implemented the program to find First and Follow of any given grammar.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

grammar.c:

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

struct ProductionRule{
    char symbol;
    char expression[20];
};

struct Grammar{
    char startState;
    char* non_terminals;
    char* terminals;
    struct ProductionRule* rules;
    int production_num;
};

struct LMDStackNode {
    struct ProductionRule rule;
    struct LMDStackNode* next;
};

struct LMDStackNode* head = NULL;

void free_grammar(struct Grammar* g){
    if (!g) return;
    if (g->non_terminals) free(g->non_terminals);
    if (g->terminals) free(g->terminals);
    if (g->rules) free(g->rules);
    free(g);
}
int find_index(char s[], char c){
    int n = strlen(s);
    for (int i=0;i<n;++i){
        if (s[i]==c){
            return i;
```

Date: 10.09.25

EXPERIMENT 3.3
RECURSIVE DESCENT PARSER

AIM:

Construct a recursive descent parser for an expression.

ALGORITHM:

Step 0: Start

Step 1: Input Grammar

- **1.1:** Input the number of non-terminals, terminals, and production rules.
- **1.2:** Input the start symbol of the grammar.
- **1.3:** Input the list of non-terminals and store them.
- **1.4:** Input the list of terminals and store them.
- **1.5:** Input each production rule in the form A->xyz.
- **1.6:** Validate each production rule:
  - **1.6.1:** Ensure the LHS symbol is a valid non-terminal.
  - **1.6.2:** Ensure RHS contains only valid terminals, non-terminals, or epsilon (e).
- **1.7:** Store the grammar in the allocated data structures (Grammar struct).

Step 2: Read the input string to be checked.

Step 3: Ensure all characters are valid grammar terminals.

Step 4: Create a derivation stack (LMDStack) to store applied production rules.

Step 5: Initialize an expanded string with the start symbol.

Step 6: Recursive Descent Parsing Procedure:

- **6.1:** If the expanded string ends (\0):
  - **6.1.1:** Compare with the input string.
  - **6.1.2:** If equal then return Accepted, else return Rejected.
- **6.2:** If the current symbol of input and expanded match:
  - **6.2.1:** Move one step ahead in both input and expanded.
  - **6.2.2:** Call recursiveDescent again.

```
        }
    }
    return -1;
}
bool str_contains(char str[],char c){
    return find_index(str,c)!=-1;
}
void add_str(char str[], char c){
    int n = strlen(str);
    for (int i=0;i<n;++i){
        if (str[i]==c){
            return ;
        }
    }
    str[n] = c;
    str[n+1] = '\0';
}
bool validTerminal(struct Grammar* g, char c){
    return str_contains(g->terminals,c);
}

bool validNonTerminal(struct Grammar* g, char c){
    return str_contains(g->non_terminals,c);
}

bool validInput(struct Grammar* g, char input[]){
    int n = strlen(input);
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i])){
            return true;
        }
    }
```

6.3: If the current expanded symbol is a non-terminal:

6.3.1: For each production rule with this non-terminal as LHS:

6.3.1.1: Save a copy of current expanded string.

6.3.1.2: Replace the non-terminal with RHS of the production.

6.3.1.3: Push this production onto derivation stack.

6.3.1.4: Call recursiveDescent with modified expanded.

6.3.1.5: If parsing succeeds → return success.

6.3.1.6: If parsing fails → pop production from stack and restore expanded.

6.4: If no matching production leads to success → return failure.

Step 7: If parsing is successful then print **“**String accepted” else print “String rejected”.

Step 8: Print Leftmost Derivation:

8.1: Traverse the derivation stack in reverse order.

8.2: Print each applied production in the order of derivation.

Step 9: Free allocated memory for non-terminals, terminals, and production rules.

Step 10: Free the grammar structure.

Step 11: Stop

```
    return true;
}

bool validExpansion(struct Grammar* g, char input[]){
    int n = strlen(input);
    if (n==1 && input[0]=='e') return true;
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i]) && !validNonTerminal(g,input[i])){
            return true;
        }
    }
    return true;
}
struct Grammar* read_grammar() {
    int num_non_terminal, num_terminal, num_production_rule;
    scanf("%d %d %d",&num_non_terminal,&num_terminal,&num_production_rule);
    struct Grammar* g = malloc(sizeof(struct Grammar));
    if (!g){
        printf("Coulnd't create grammar\n");
        return NULL;
    }
    scanf(" %c",&g->startState);
    if (g->startState==EOF){
        printf("Reached EOF when reading start state\n");
        free_grammar(g);
        return NULL;
    }
    g->production_num = num_production_rule;
    //Read non terminals
    g->non_terminals = malloc(sizeof(char)*num_non_terminal);
    if (!g->non_terminals){
```

```
        printf("Couldnt' allocate non terminals\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_non_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading non terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->non_terminals[i] = c;
    }
    g->non_terminals[num_non_terminal] = '\0';
    //Read terminals
    g->terminals = malloc(sizeof(char)*num_terminal);
    for (int i=0;i<num_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->terminals[i] = c;
    }
    g->terminals[num_terminal] = '\0';

    //Read Production Rules
}
```

```
    g->rules = malloc(sizeof(struct ProductionRule)*num_production_rule);
    if (!g){
        printf("Error reading production rules\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_production_rule;++i){
        char rule[20];
        scanf("%s",rule);
        sscanf(rule,"%c->%s",&(g->rules[i].symbol),&g->rules[i].expression);
        if (!validNonTerminal(g,g->rules[i].symbol) || !validExpansion(g,g-
>rules[i].expression)){
            printf("Production rule %s invalid\n",rule);
            if (!validNonTerminal(g,g->rules[i].symbol)){
                printf("Invalid symbol on LHS\n");
            }
            if (!validExpansion(g,g->rules[i].expression)){
                printf("Invalid expression on RHS");
            }
            free_grammar(g);
            return NULL;
        }}
    return g;
}
void push_derivation(struct ProductionRule r){
    struct LMDStackNode* n = malloc(sizeof(struct LMDStackNode));
    n->next = head;
    n->rule = r;
    head = n;
}
bool empty_derivation(){
```

```
if (head) return false;
    return true;
}
void pop_derivation(){
    if (!head) return;
    struct LMDStackNode* n = head->next;
    free(head);
    head = n;
}
struct ProductionRule top_derivation(){
    return head->rule;
}
void print_delete_derivation(){
    if (empty_derivation()) return;
    struct ProductionRule p = top_derivation();
    pop_derivation();
    print_delete_derivation();
    printf("%c->%s\n",p.symbol,p.expression);
    printf("\n");
}
```

recursive_descent.c

```
#include "grammar.c"
int recursiveDescent(struct Grammar* g, char input[],int inputStart, char expanded[],
int expandedStart){
    if (expanded[expandedStart]=='\0'){
        return strcmp(input,expanded)==0;
    }

    if (input[inputStart]==expanded[expandedStart]){
        return recursiveDescent(g, input, inputStart + 1, expanded, expandedStart + 1);
    }

    char current = expanded[expandedStart];
    for (int i=0;i<g->production_num;++i){
```

```
    if (current==g->rules[i].symbol){
            char expanded_copy[100];
            strcpy(expanded_copy,expanded);

            expanded[expandedStart] = '\0';
            if (g->rules[i].expression[0]!='e' || g->rules[i].expression[1]!='\0'){
                strcat(expanded,g->rules[i].expression);
            }
            strcat(expanded,expanded_copy+expandedStart+1);
            push_derivation(g->rules[i]);

            if (recursiveDescent(g,input,inputStart,expanded,expandedStart)) {
                return true;
            }

            pop_derivation();
            strcpy(expanded,expanded_copy);
        }
    }
    return false;
}

bool parse(struct Grammar* g,char input[]){
    char expanded[100];
    expanded[0] = g->startState;
    expanded[1] = '\0';
    return recursiveDescent(g,input,0,expanded,0);
}

int main(){
    struct Grammar* g = read_grammar();
    char input[20];
    scanf("%s",input);
    if (parse(g,input)){
        printf("String accepted\n");
    } else {
        printf("String rejected\n");
    }
    free(g);
    print_delete_derivation();
}
```

input.txt:

2 2 3
E
EZ
+i
E->iZ
Z->+iZ
Z->e
i+i+i

OUTPUT:

String accepted
E->iZ
Z->+iZ
Z->+iZ
Z->e

RESULT:

Implemented the program for a recursive descent parser.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

grammar.c:

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

struct ProductionRule{
    char symbol;
    char expression[20];
};

struct Grammar{
    char startState;
    char* non_terminals;
    char* terminals;
    struct ProductionRule* rules;
    int production_num;
};

struct LMDStackNode {
    struct ProductionRule rule;
    struct LMDStackNode* next;
};

struct LMDStackNode* head = NULL;

void free_grammar(struct Grammar* g){
    if (!g) return;
    if (g->non_terminals) free(g->non_terminals);
    if (g->terminals) free(g->terminals);
    if (g->rules) free(g->rules);
    free(g);
}
int find_index(char s[], char c){
    int n = strlen(s);
    for (int i=0;i<n;++i){
        if (s[i]==c){
            return i;
```

Date: 18.09.25

EXPERIMENT 3.4
SHIFT REDUCE PARSER

AIM:

Construct a shift reduce parser for a given language.

ALGORITHM:

Step 0: Start

Step 1: Input Grammar

- **1.1:** Input the number of non-terminals, terminals, and production rules.
- **1.2:** Input the start symbol of the grammar.
- **1.3:** Input the list of non-terminals and store them in memory.
- **1.4:** Input the list of terminals and store them in memory.
- **1.5:** Input each production rule in the form A->xyz.
- **1.6:** Validate each rule:
  - **1.6.1:** Check if LHS is a valid non-terminal.
  - **1.6.2:** Check if RHS is composed of valid terminals/non-terminals/epsilon.
- **1.7:** Store rules into the Grammar structure.

Step 2: Read the string to be parsed.

Step 3: Check that each symbol of the input belongs to the grammar's terminals.

Step 4: If invalid symbol found then print error and stop.

Step 5: Initialize Stacks

- **5.1:** Create two stacks:
  - **5.1.1:** Input Stack that holds the input string symbols
  - **5.1.2:** Output Stack that is initially empty, used for shift/reduce operations.
- **5.2:** Create a derivation stack to keep track of applied productions (for printing derivation later).

```
        }
    }
    return -1;
}
bool str_contains(char str[],char c){
    return find_index(str,c)!=-1;
}
void add_str(char str[], char c){
    int n = strlen(str);
    for (int i=0;i<n;++i){
        if (str[i]==c){
            return ;
        }
    }
    str[n] = c;
    str[n+1] = '\0';
}
bool validTerminal(struct Grammar* g, char c){
    return str_contains(g->terminals,c);
}

bool validNonTerminal(struct Grammar* g, char c){
    return str_contains(g->non_terminals,c);
}

bool validInput(struct Grammar* g, char input[]){
    int n = strlen(input);
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i])){
            return true;
        }
    }
```

Step 6: P Repeat until the string is either accepted or rejected:

6.1: For each production rule A->α in grammar:

6.1.1: Compare RHS α with top symbols of the Output Stack.

6.1.2: If they match:

6.1.1.1: Pop |α| symbols from the Output Stack.

6.1.1.2: Push LHS non-terminal A onto Output Stack.

6.1.1.3: Record this production on the derivation stack.

6.1.1.4: Print action as Reduce with current states.

6.1.1.5: Restart parsing loop from Step 4.

6.2: If Input Stack is not empty:

6.2.1: Pop the top symbol from Input Stack.

6.2.2: Push it onto Output Stack.

6.2.3: Print action as Shift with current states.

6.2.4: Restart parsing loop from Step 4.

6.3 : If Input Stack is empty and Output Stack has exactly one symbol equal to Start Symbol then Print “String Accepted” else print “String Rejected”.

Step 7: Traverse the derivation stack and print each production applied which gives the Rightmost Derivation (RMD) sequence of the input.

Step 8: Free memory.

Step 9: Stop

```
    return true;
}

bool validExpansion(struct Grammar* g, char input[]){
    int n = strlen(input);
    if (n==1 && input[0]=='e') return true;
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i]) && !validNonTerminal(g,input[i])){
            return true;
        }
    }
    return true;
}
struct Grammar* read_grammar() {
    int num_non_terminal, num_terminal, num_production_rule;
    scanf("%d %d %d",&num_non_terminal,&num_terminal,&num_production_rule);
    struct Grammar* g = malloc(sizeof(struct Grammar));
    if (!g){
        printf("Coulnd't create grammar\n");
        return NULL;
    }
    scanf(" %c",&g->startState);
    if (g->startState==EOF){
        printf("Reached EOF when reading start state\n");
        free_grammar(g);
        return NULL;
    }
    g->production_num = num_production_rule;
    //Read non terminals
    g->non_terminals = malloc(sizeof(char)*num_non_terminal);
    if (!g->non_terminals){
```

```
        printf("Couldnt' allocate non terminals\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_non_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading non terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->non_terminals[i] = c;
    }
    g->non_terminals[num_non_terminal] = '\0';
    //Read terminals
    g->terminals = malloc(sizeof(char)*num_terminal);
    for (int i=0;i<num_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->terminals[i] = c;
    }
    g->terminals[num_terminal] = '\0';

    //Read Production Rules
}
```

```
    g->rules = malloc(sizeof(struct ProductionRule)*num_production_rule);
    if (!g){
        printf("Error reading production rules\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_production_rule;++i){
        char rule[20];
        scanf("%s",rule);
        sscanf(rule,"%c->%s",&(g->rules[i].symbol),&g->rules[i].expression);
        if (!validNonTerminal(g,g->rules[i].symbol) || !validExpansion(g,g-
>rules[i].expression)){
            printf("Production rule %s invalid\n",rule);
            if (!validNonTerminal(g,g->rules[i].symbol)){
                printf("Invalid symbol on LHS\n");
            }
            if (!validExpansion(g,g->rules[i].expression)){
                printf("Invalid expression on RHS");
            }
            free_grammar(g);
            return NULL;
        }}
    return g;
}
void push_derivation(struct ProductionRule r){
    struct LMDStackNode* n = malloc(sizeof(struct LMDStackNode));
    n->next = head;
    n->rule = r;
    head = n;
}
bool empty_derivation(){
```

```
if (head) return false;
    return true;
}
void pop_derivation(){
    if (!head) return;
    struct LMDStackNode* n = head->next;
    free(head);
    head = n;
}
struct ProductionRule top_derivation(){
    return head->rule;
}
void print_delete_derivation(){
    if (empty_derivation()) return;
    struct ProductionRule p = top_derivation();
    pop_derivation();
    print_delete_derivation();
    printf("%c->%s\n",p.symbol,p.expression);
    printf("\n");
        }
```

stack.c:

```
#include "grammar.c"
struct StackNode{
    char symbol;
    char firstTerminal;
    struct StackNode* next;
};


bool emptyStack(struct StackNode** indirect){
    if (*indirect) return false;
    return true;
}
```

```
void stackPush(struct StackNode** indirect,char symbol, bool terminal){
    char firstTerminal = '$';
    if (terminal){
        firstTerminal = symbol;
    } else if (*indirect){
        firstTerminal = (*indirect)->firstTerminal;
    }
    struct StackNode* st = malloc(sizeof(struct StackNode));
    st->symbol = symbol;
    st->firstTerminal = firstTerminal;
    st->next = *indirect;
    *indirect = st;
}

void popStack(struct StackNode** indirect){
    if (*indirect){
        struct StackNode* st = *indirect;
        *indirect = st->next;
        free(st);
    }
}

char stackTopValue(struct StackNode** indirect){
    if (*indirect){
        return (*indirect)->symbol;
    }
    return '$';
}
char stackTerminal(struct StackNode** indirect){
    if (!emptyStack(indirect)){
        return (*indirect)->firstTerminal;
    }
    return '$';
}
void freeStack(struct StackNode** indirect){
    while (!emptyStack(indirect)){
        popStack(indirect);
    }
}

void printState(struct StackNode** indirect){
```

```
while (*indirect){
        printf("%c",(*indirect)->symbol);
        // printf("(%c,%c)",(*indirect)->symbol,(*indirect)->firstTerminal);
        indirect = &((*indirect)->next);
    }
    printf("$");
}
```

shift_reduce_common.c:

```
#include "stack.c"
bool match(struct StackNode** indirect, char s[]){
    int n = strlen(s);
    struct StackNode* current = *indirect;

    for (int i=n-1;i>=0;--i){
        if (!current){
            return false;
        }
        char lhs = current->symbol;
        char rhs = s[i];
        if (lhs!=rhs){
            return false;
        }

        current = current->next;
    }

    // Don't remove matched nodes here, that happens in reduce()

    return true;
}

bool shift(struct StackNode** inputStack,struct StackNode** outputStack){
    if (!emptyStack(inputStack)){
        stackPush(outputStack,stackTopValue(inputStack),true);
        popStack(inputStack);
        printf("Action: Shift Input: ");
        printState(inputStack);
        printf(" Output: ");
        printState(outputStack);
        printf("\n");
        return true;
    }
```

```
    return false;
}

bool reduce(struct StackNode** outputStack,struct StackNode** inputStack,struct
Grammar* g){
    int np = g->production_num;
    bool res = false;
    for (int p=0;p<np;++p){
        char* expression = g->rules[p].expression;
        char symbol = g->rules[p].symbol;
        if (match(outputStack,expression)){
            int n = strlen(expression);
            while (n-->0){
                popStack(outputStack);
            }
            stackPush(outputStack,symbol,false);
            push_derivation(g->rules[p]);
            res = true;
            printf("Action: Reduce Input: ");
            printState(inputStack);
            printf(" Output: ");
            printState(outputStack);
            printf("\n");
        }
    }
    return res;
}
void derivation_parse(){
    printf("The RMD is as follows:\n");
    while (!empty_derivation()){
        struct ProductionRule r = top_derivation();
        printf("%c->%s\n",r.symbol,r.expression);
        pop_derivation();
    }
}
```

shift_reduce_parse.c:

```
#include "shift_reduce_common.c"
void parse(struct Grammar* g,char input[20]){
    struct StackNode * inputHead = NULL;
    struct StackNode * stackHead = NULL;

    struct StackNode** inputStack = &inputHead;
```

```
struct StackNode** outputStack = &stackHead;
    int n = strlen(input);
    for (int i=n-1;i>=0;--i){
        stackPush(inputStack,input[i],true);
    }
    int max_iterations = 1000;
    while (max_iterations-->0){
        //Try reduce
        if (reduce(outputStack,inputStack,g)){
            continue;
        }

        //Try shift
        if (shift(inputStack,outputStack)){
            continue;
        }

        //Accept or reject if neither works
        if (emptyStack(inputStack) && !emptyStack(outputStack)
        && !stackHead->next && stackTopValue(outputStack)==g->startState){
            //Valid input
            printf("String Accepted\n");
        } else {
            printf("String rejected\n");
        }
        break;
    }
    derivation_parse();
    freeStack(inputStack);
    freeStack(outputStack);
}

int main(){
    struct Grammar* g = read_grammar();
    int n = strlen(g->terminals)+1;
    char input[20];
    scanf("%19s",input);
    if (validInput(g,input)){
        parse(g,input);
    } else {
        printf("Invalid input\n");
    }
    return 0;
}
```

input.txt:

1 3 3
E
E
i+*
E->E+E
E->E*E
E->i
i+i*i

OUTPUT:

Action: Shift Input: +i*i$ Output: i$
Action: Reduce Input: +i*i$ Output: E$
Action: Shift Input: i*i$ Output: +E$
Action: Shift Input: *i$ Output: i+E$
Action: Reduce Input: *i$ Output: E+E$
Action: Reduce Input: *i$ Output: E$
Action: Shift Input: i$ Output: *E$
Action: Shift Input: $ Output: i*E$
Action: Reduce Input: $ Output: E*E$
Action: Reduce Input: $ Output: E$

String Accepted

The RMD is as follows:
E->E*E
E->i
E->E+E
E->i
E->i

RESULT:

Successfully implemented the program for a shift reduce parser.

Name: Aravind Ashokan
Roll No: 16

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
void gen_code_for_operator(char *inp, char operator, char *reg) {
    int i = 0, j = 0;char temp[100];
    while (inp[i] != '\0') {
        if (inp[i] == operator) {
            if (operator == '=') {
                printf("%c\t%c\t%c\t--\n", operator, inp[i - 1], inp[i + 1]);
                temp[j - 1] = inp[i - 1];
            } else {
                printf("%c\t%c\t%c\t%c\n", operator, *reg, inp[i - 1], inp[i + 1]);
                temp[j - 1] = *reg;     (*reg)--;
            }  i += 2;    continue;
        } temp[j] = inp[i];     i++;   j++;
    }  temp[j] = '\0';
    strcpy(inp, temp);
}void gen_code(char *inp) {
    char reg = 'Z';
    gen_code_for_operator(inp, '/', &reg);
    gen_code_for_operator(inp, '*', &reg);
    gen_code_for_operator(inp, '+', &reg);
    gen_code_for_operator(inp, '-', &reg);
    gen_code_for_operator(inp, '=', &reg);
}void main() {
    char inp[100];
    printf("Enter expression: ");
    scanf("%s", inp);
    printf("Oprtr\tDestn\tOp1\tOp2\n");
    gen_code(inp);
}
```

OUTPUT:

Enter the expression: q=a-b/c+d*e

Oprtr	Destn	Op1	Op2
/	Z	b	c
*	Y	d	e
+	X	Z	Y
-	W	a	X
=	q	W	--

Date: 18.09.2025

EXPERIMENT- 4.1

INTERMEDIATE CODE GENERATION

AIM:

Implement Intermediate code generation for simple expressions.

ALGORITHM:

Step 1: Start
Step 2: Input & Initialization
- Prompt the user to enter an arithmetic expression (e.g., a=b+c*d)
- Store the input in inp[]
- Initialize a register variable reg = 'Z' for naming intermediate results

Step 3: Display Table Header
- Print column headers: Oprtr, Destn, Op1, Op2

Step 4: Generate Code for Each Operator by Precedence
- For each operator in the order: /, *, +, -, =
  - Scan the string inp[] from left to right
  - If the current character matches the operator:
    - If operator is '=':
      - Set destination as the left-hand side (LHS)
      - Set operand 1 as the right-hand side (RHS)
      - Set operand 2 as --
      - Print the line in the format: = LHS RHS --
    - Else (for all other operators):
      - Set destination as current register
      - Set operand 1 as the symbol before the operator
      - Set operand 2 as the symbol after the operator
      - Print the line in the format: op reg op1 op2
      - Replace the subexpression in inp[] with the current register
      - Decrement reg to get the next temporary register

Step 5: Stop.

RESULT:

Implemented Intermediate code generation for simple expressions successfully.