HMR INSTITUTE OF TECHNOLOGY AND MANAGEMENT

Hamidpur, Delhi-110036

(Affiliated to Guru Gobind Singh Indraprastha University, New Delhi)

Submitted in partial fulfilment of the requirements

For the award of degree of

Bachelor of Technology

In

Computer Science and Engineering

2021-2025

COMPILER DESIGN

Lab File

Code: CIC-351

SUBMITTED TO:

Mr. Akshay Kalyan

Assistant Professor

CSE Department

SUBMITTED BY:

Ashwani Shivam

01196507222

CSE 5-C

INDEX

S.No	Name of Program	Date	Sign	Remarks
1.	Practiceof LEX/YACC of Compiler Writing.			
2.	Write a program to check whether a string belong to the grammar or not.			
3.	Write a program to check whether a string include keyword or not.			
4.	Write a program to remove left recursion from Grammar.			
5.	Write a program to perform Left Factoring on a Grammar.			
6.	Write a program to show all the operations of a stack.			
7.	Write a program to find out the leading of the non-terminal in a grammar.			
8.	Write a program to find out the FIRST of the Non-Terminals in a grammar			
9.	Write a program to check whether a grammar is operator precedent.			
10.	Write a program to implement Shift Reduce parsing for a String.			

PRACTICAL 1

AIM: Practiceof LEX/YACC of Compiler Writing.

THEORY:

A compiler or interpreter for a programming language is often decomposed into two parts:
1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.
Lex and Yacc can generate program fragments that solve the first task. The task of discovering the source structure again is decomposed into subtasks:
1. Split the source file into tokens (Lex).
2. Find the hierarchical structure of the program (Yacc).
Lex - A Lexical Analyzer Generator Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine. Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it. Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7.

Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system. Lex generates programs to be used in simple lexical analysis of text. The input files (standard input default) contain regular expressions to be searched for and actions written in C to be executed when expressions are found. A C source program, lex.yy.c is generated. This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action foreach regular expression that is recognized. The options have the following meanings. Place the result on the standard output instead of in file

lex.yy.c. –v Print a one–line summary of statistics of the generated analyzer. –n Opposite of –v; –n is default. –9 Adds code to be able to compile through the native C compilers.

EXAMPLE This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%% [A–Z]
putchar(yytext[0]+'a'–'A');
        [ ]+$ [ ]+ putchar(' ')
```

Yacc: Yet Another Compiler-Compiler

Yacc is a computer program that serves as the standard parser generator on Unix systems. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser (the part of a compiler that tries to make sense of the input) based on an analytic grammar written in BNF notation. Yacc generates the code for the parser in the C programming language. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the usersupplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. The input subroutine produced by Yacc calls a usersupplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules. In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Yacc converts a context–free grammar and translation code into a set of tables for an LR(1) parser and translator. The grammar may be ambiguous; specified precedence rules are used to break ambiguities. The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with a lexical analyzer function, yylex(void) (often generated by lex(1)), with a main(int argc, char *argv[]) program, and with an error handling routine, yyerror(char*). The options are –o output Direct output to the specified file instead of y.tab.c. Create file y.debug, containing diagnostic messages. v Create file y.output, containing a description of the parsing tables and of conflicts arising from ambiguities in the grammar. –d Create file y.tab.h, containing #define statements that associate yacc– assigned `token codes' with user–declared `token names'. Include it in source files other than y.tab.c to give access to the token codes. –s stem Change the prefix y of the file names y.tab.c,y.tab.h,

y.debug, and y.output to stem. –S Write a parser that uses Stdio instead of the print routines in libc.

PRACTICAL 2

AIM: Write a program to check whether a string belong to the grammar or not.

Program:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main() {
   char string[50];
   int flag, count = 0;
clrscr();
printf("The grammar is: S->aS, S->Sb, S->ab\n");
printf("Enter the string to be checked:\n");
   gets(string);
   if (string[0] == 'a') {
      flag = 0;
      for (count = 1; string[count - 1] != '\0'; count++) {
         if (string[count] == 'b') {
            flag = 1;
            continue;
         } else if ((flag == 1) && (string[count] == 'a')) {
printf("The string does not belong to the specified grammar");
            break;
         } else if (string[count] == 'a') {
            continue;
         } else if ((flag == 1) && (string[count] == '\0')) {
printf("String accepted…..!!!!");
            break;
         } else {
printf("String not accepted");
         }
```

```
        }
    }
getch();
}
```

OUTPUT:

```
 The grammer is:
  S->aS
  S->Sb
  S->ab
Enter the string to be checked:
aab
String accepted....!!!!
```

PRACTICAL 3

AIM: Write a program to check whether a string include keyword or not.

Program:

```
#include <stdio.h>
#include <string.h>
int check_keyword(char* string, char* keyword) {
    if (strstr(string, keyword) != NULL) {
        return 1;
    } else {
        return 0;
    }
}
int main() {
    char string[100];
    char keyword[20];
printf("Enter a string: ");
fgets(string, sizeof(string), stdin);
printf("Enter a keyword: ");
fgets(keyword, sizeof(keyword), stdin);
    // Removing newline characters from input
    string[strcspn(string, "\n")] = '\0';
    keyword[strcspn(keyword, "\n")] = '\0';
    int result = check_keyword(string, keyword);
    if (result) {
printf("The string includes the keyword.\n");
    } else {
printf("The string does not include the keyword.\n");
    }
    return 0;
}
```

OUTPUT:

```
PS C:\Users\user\Code of Cpp> cd "c:\Users\user\Code of Cpp\" ; if ($?) { gcc Compile
rDesign.c -o CompilerDesign } ; if ($?) { .\CompilerDesign }
Enter a string: ABABABAB
Enter a keyword: ABA
The string includes the keyword.
PS C:\Users\user\Code of Cpp> cd "c:\Users\user\Code of Cpp\" ; if ($?) { gcc Compile
rDesign.c -o CompilerDesign } ; if ($?) { .\CompilerDesign }
Enter a string: ABBA
Enter a keyword: ABA
The string does not include the keyword.
PS C:\Users\user\Code of Cpp>
```

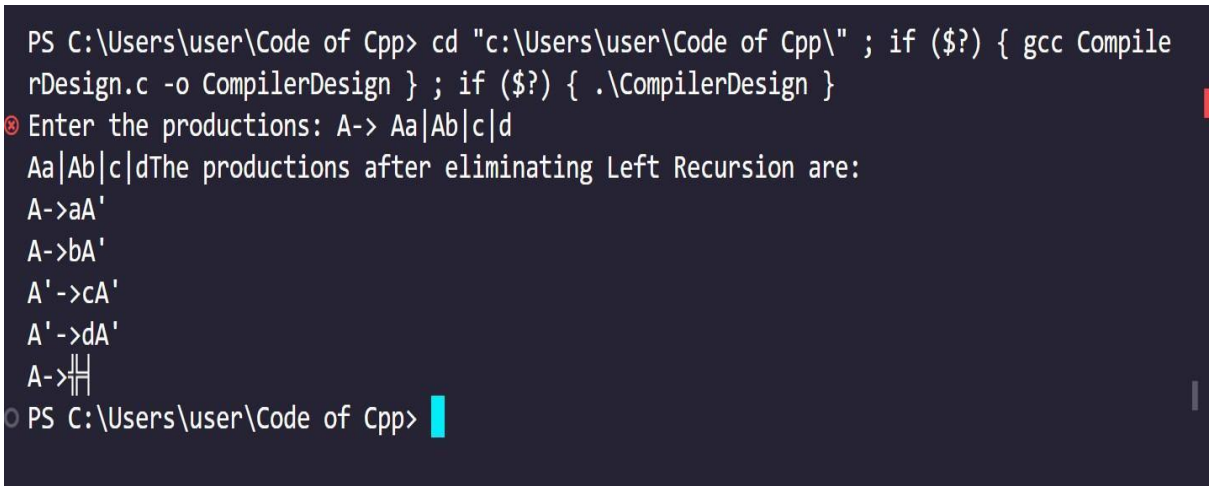
PRACTICAL 4

AIM: Write a program to remove left recursion from Grammar.

Program:

```
#include <stdio.h>
#include <string.h>
void main() {
   char input[100], l[50], r[50], temp[10], tempprod[20], productions[25][50];
   int i = 0, j = 0, flag = 0, consumed = 0;
printf("Enter the productions: ");
scanf("%1s->%s", l, r);
printf("%s", r);
   while (sscanf(r + consumed, "%[^|]s", temp) == 1 && consumed <= strlen(r)) {
      if (temp[0] == l[0]) {
         flag = 1;
sprintf(productions[i++], "%s->%s%s'\0", l, temp + 1, l);
      } else {
sprintf(productions[i++], "%s'->%s%s'\0", l, temp, l);
      }
      consumed += strlen(temp) + 1;
   }
   if (flag == 1) {
sprintf(productions[i++], "%s->ε\0", l);
printf("The productions after eliminating Left Recursion are:\n");
      for (j = 0; j <i; j++)
printf("%s\n", productions[j]);
   } else {
printf("The Given Grammar has no Left Recursion");
   }
}
```

OUTPUT:


PS C:\Users\user\Code of Cpp> cd "c:\Users\user\Code of Cpp\" ; if ($?) { gcc Compile
rDesign.c -o CompilerDesign } ; if ($?) { .\CompilerDesign }
Enter the productions: A-> Aa|Ab|c|d
Aa|Ab|c|dThe productions after eliminating Left Recursion are:
A->aA'
A->bA'
A'->cA'
A'->dA'
A->ε
PS C:\Users\user\Code of Cpp>

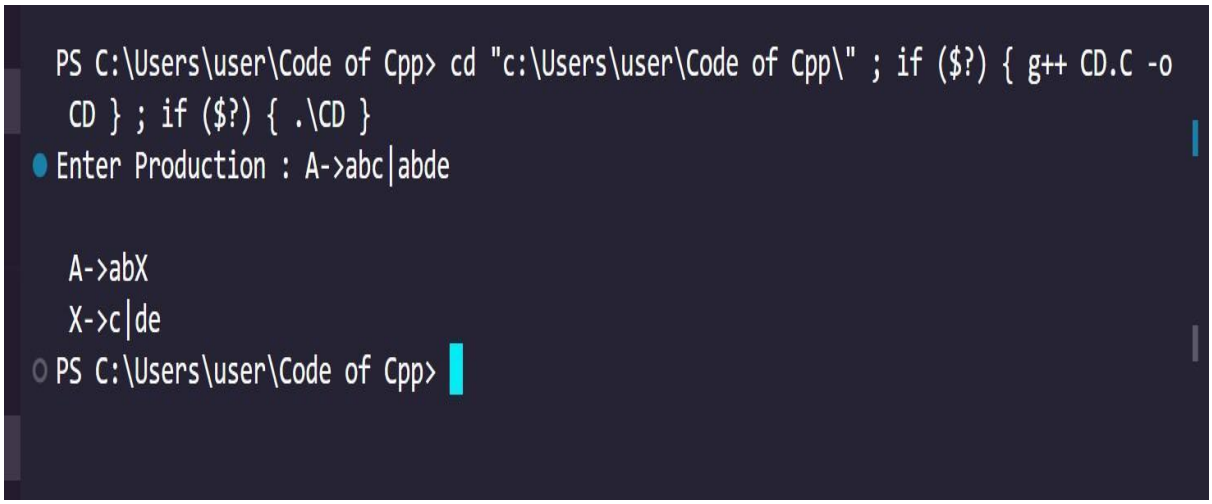
PRACTICAL 5

AIM: Write a program to perform Left Factoring on a Grammar.

Program:

```
#include<stdio.h>
#include<string.h>
int main() {
   char gram[20], part1[20], part2[20], modifiedGram[20], newGram[20], tempGram[20];
   int i, j = 0, k = 0, l = 0, pos;
printf("Enter Production : A->");
   gets(gram);
   // Separate the two parts of the production
   for (i = 0; gram[i] != '|'; i++, j++)
      part1[j] = gram[i];
part1[j] = '\0';
   for (j = ++i, i = 0; gram[j] != '\0'; j++, i++)
      part2[i] = gram[j];
part2[i] = '\0';
   for (i = 0; i<strlen(part1) || i<strlen(part2); i++) {
      if (part1[i] == part2[i]) {
modifiedGram[k] = part1[i];
         k++;
pos = i + 1;
      }
   }
   for (i = pos, j = 0; part1[i] != '\0'; i++, j++)
newGram[j] = part1[i];
newGram[j++] = '|';
   for (i = pos; part2[i] != '\0'; i++, j++)
newGram[j] = part2[i];
modifiedGram[k] = 'X';
      modifiedGram[++k] = '\0';
      newGram[j] = '\0';
      printf("\n A->%s", modifiedGram);
      printf("\n X->%s\n", newGram);
   return 0;
}
```

OUTPUT:


PS C:\Users\user\Code of Cpp> cd "c:\Users\user\Code of Cpp\" ; if ($?) { g++ CD.C -o CD } ; if ($?) { .\CD }
Enter Production : A->abc|abde
A->abX
X->c|de
PS C:\Users\user\Code of Cpp>

PRACTICAL 6

AIM: Write a program to show all the operations of a stack.

Program:

```
#include<stdio.h>
int stack[10], choice, n, top, x, i; // Declaration of variables
void push();
void pop();
void display();
int main() {
   top = -1; // Initially there is no element in stack
printf("\n Enter the size of STACK : ");
scanf("%d", &n);
printf("\nSTACK IMPLEMENTATION USING ARRAYS\n");
   do {
printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n");
printf("\nEnter the choice : ");
scanf("%d", &choice);
       switch (choice) {
          case 1: { push(); break; }
          case 2: { pop(); break; }
          case 3: { display(); break; }
          case 4: { break; }
          default: { printf("\nInvalid Choice\n"); }
       }
   } while (choice != 4);
   return 0;
}
void push() {
   if (top >= n - 1) {
printf("\nSTACK OVERFLOW\n");
   } else {
printf("Enter a value to be pushed : ");
scanf("%d", &x);
       top++; // TOP is incremented after an element is pushed
       stack[top] = x; // The pushed element is made as TOP
   }
}
void pop() {
   if (top <= -1) {
printf("\nSTACK UNDERFLOW\n");
   } else {
printf("\nThe popped element is %d", stack[top]);
       top--; // Decrement TOP after a pop
   }
}
void display() {
   if (top >= 0) {
```

```
        // Print the stack
printf("\nELEMENTS IN THE STACK\n\n");
        for (i = top; i>= 0; i--)
printf("%d\t", stack[i]);
    } else {
printf("\nEMPTY STACK\n");
    }
}
```

OUTPUT:

```
 Enter the size of STACK : 5


STACK IMPLEMENTATION USING ARRAYS

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 1
Enter a value to be pushed : 5

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 1
Enter a value to be pushed : 10

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 1
Enter a value to be pushed : 15

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 1
Enter a value to be pushed : 20

1.PUSH
```

```
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 1
Enter a value to be pushed : 25

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 3

ELEMENTS IN THE STACK

25      20      15      10      5
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 2

The popped element is 25
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 2

The popped element is 20
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice : 3

ELEMENTS IN THE STACK

15      10      5
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the choice :
```

PRACTICAL 7

AIM: Write a program to find out the leading of the non-terminal in a grammar.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_PROD 10
#define MAX_LEN 10
// Function to calculate the leading set for a given non-terminal
voidcalculateLeading(charnonTerminal,char grammar[][MAX_LEN], int
numProductions, char leadingSet[]) {
   int i, j;
   for (i = 0; i<numProductions; i++) {
      if (grammar[i][0] == nonTerminal) {
         // If production starts with the non-terminal
         if (grammar[i][3] >= 'a' && grammar[i][3] <= 'z') {
            // If the first symbol is a terminal
leadingSet[grammar[i][3] - 'a'] = 1;
         } else if (grammar[i][3] >= 'A' && grammar[i][3] <= 'Z') {
            // If the first symbol is a non-terminal, calculate its leading set
            for (j = 0; j <numProductions; j++) {
               if (grammar[j][0] == grammar[i][3]) {
calculateLeading(grammar[i][3], grammar, numProductions, leadingSet);
               }
            }
         }
      }
   }
}

// Function to display the leading set for each non-terminal
void displayLeadingSets(char nonTerminals[], char grammar[][MAX_LEN], int
numProductions) {
   int i, j;
printf("\nLeading Sets:\n");
   for (i = 0; i<strlen(nonTerminals); i++) {
      char leadingSet[MAX_LEN] = {0};
calculateLeading(nonTerminals[i], grammar, numProductions, leadingSet);
printf("Leading(%c) = {", nonTerminals[i]);
      for (j = 0; j < 26; j++) {
         if (leadingSet[j] == 1) {
printf("%c, ", 'a' + j);
         }
      }
printf("}\n");
   }
}
```

```
int main() {
    char nonTerminals[MAX_LEN];
    char grammar[MAX_PROD][MAX_LEN];
    int numProductions;
printf("Enter the number of productions: ");
scanf("%d", &numProductions);
printf("Enter the non-terminals: ");
scanf("%s", nonTerminals);
printf("Enter the productions:\n");
    for (int i = 0; i<numProductions; i++) {
scanf("%s", grammar[i]);
    }
displayLeadingSets(nonTerminals, grammar, numProductions);
    return 0;
}
```

OUTPUT:

```
PS C:\Users\user\Code of Cpp> cd "c:\Users\user\Code of Cpp\" ; if ($?) { gcc Compile
rDesign.c -o CompilerDesign } ; if ($?) { .\CompilerDesign }
Enter the number of productions: 3
Enter the non-terminals: SAB
Enter the productions:
S->AB
A->aA|epsilon
B->bB|c

Leading Sets:
Leading(S) = {a, x, }
Leading(A) = {a, o, x, }
Leading(B) = {b, x, }
PS C:\Users\user\Code of Cpp>
```

PRACTICAL 8

AIM: Write a program to find out the FIRST of the Non-Terminals in a Grammar.

Program:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main() {
   char t[5], nt[10], p[5][5], first[5][5], temp;
   int i, j, not, nont, k = 0, f = 0;
clrscr();
printf("\nEnter the no. of Non-terminals in the grammar:");
scanf("%d", &nont);
printf("\nEnter the Non-terminals in the grammar:\n");
   for (i = 0; i<nont; i++) {
scanf("\n%c", &nt[i]);
   }
printf("\nEnter the no. of Terminals in the grammar: ( Enter e for epsilon ) ");
scanf("%d", &not);
printf("\nEnter the Terminals in the grammar:\n");
   for (i = 0; i< not || t[i] == '$'; i++) {
scanf("\n%c", &t[i]);
   }
   for (i = 0; i<nont; i++) {
      p[i][0] = nt[i];
      first[i][0] = nt[i];
   }
printf("\nEnter the productions:\n");
   for (i = 0; i<nont; i++) {
scanf("%c", &temp);
printf("\nEnter the production for %c ( End the production with '$' sign ) :", p[i][0]);
      for (j = 0; p[i][j] != '$';) {
         j += 1;
scanf("%c", &p[i][j]);
      }
   }
   for (i = 0; i<nont; i++) {
printf("\nThe production for %c -> ", p[i][0]);
      for (j = 1; p[i][j] != '$'; j++) {
printf("%c", p[i][j]);
      }
   }
   for (i = 0; i<nont; i++) {
      f = 0;
      for (j = 1; p[i][j] != '$'; j++) {
         for (k = 0; k < not; k++) {
            if (f == 1)
               break;
```

```
            if (p[i][j] == t[k]) {
                first[i][j] = t[k];
                first[i][j + 1] = '$';
                f = 1;
                break;
            } else if (p[i][j] == nt[k]) {
                first[i][j] = first[k][j];
                if (first[i][j] == 'e')
                    continue;
                first[i][j + 1] = '$';
                f = 1;
                break;
            }
        }
    }
  }
  for (i = 0; i<nont; i++) {
printf("\n\nThe first of %c -> ", first[i][0]);
    for (j = 1; first[i][j] != '$'; j++) {
printf("%c\t", first[i][j]);
    }
  }
getch();
}
```

OUTPUT:

```
Enter the no. of Non-terminals in the grammer:3

Enter the Non-terminals in the grammer:
E T V

Enter the no. of Terminals in the grammer: ( Enter e for absiline ) 5

Enter the Terminals in the grammer:
+ * ( ) i

Enter the productions :

Enter the production for E ( End the production with '$' sign ) :(i)$

Enter the production for T ( End the production with '$' sign ) :i*E$

Enter the production for V ( End the production with '$' sign ) :E+i$

The production for E -> (i)
The production for T -> i*E
The production for V -> E+i

The first of E -> (

The first of T -> i

The first of V -> (
```

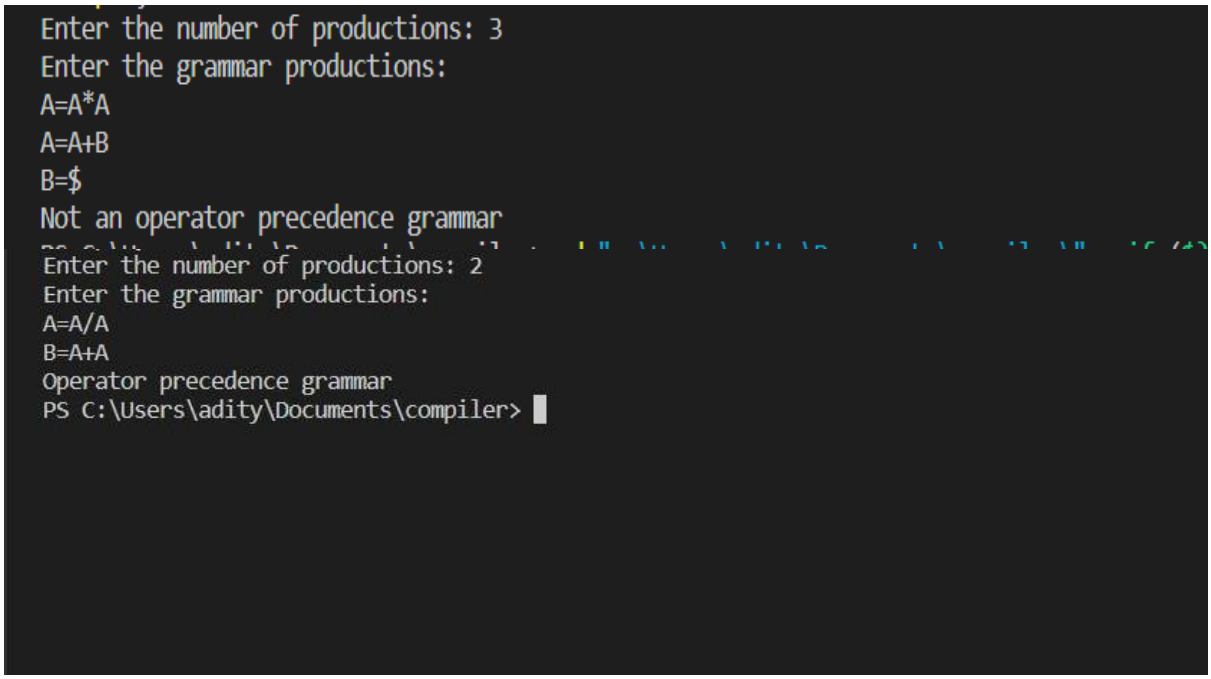
PRACTICAL 9

AIM: Write a program to check whether a grammar is operator precedent.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void exitNotOperatorPrecedence() {
printf("Not an operator precedence grammar\n");
   exit(0);
}
int main() {
   char grammar[20][20], symbol;
   int numProductions, i, j = 2, flag = 1;
// Assuming it's an operator precedence grammar initially
printf("Enter the number of productions: ");
scanf("%d", &numProductions);
printf("Enter the grammar productions:\n");
   for (i = 0; i<numProductions; i++)
scanf("%s", grammar[i]);
   for (i = 0; i<numProductions; i++) {
      symbol = grammar[i][2];
      while (symbol != '0') {
         // Check for operators
         if (grammar[i][3] == '+' || grammar[i][3] == '-' || grammar[i][3] == '*' ||
grammar[i][3] == '/') {
            // If an operator is found
            if (symbol == '+' || symbol == '-') {
               // Check for multiplication or division preceding addition or subtraction
               if (grammar[i][j + 1] == '*' || grammar[i][j + 1] == '/')
exitNotOperatorPrecedence();
            }
         } else {
            flag = 0; // Not an operator precedence grammar
exitNotOperatorPrecedence();
         }
         symbol = grammar[i][++j];
      }
   }
   if (flag)
printf("Operator precedence grammar\n");
   return 0;
}
```

OUTPUT:


Enter the number of productions: 3
Enter the grammar productions:
A=A*A
A=A+B
B=$
Not an operator precedence grammar
Enter the number of productions: 2
Enter the grammar productions:
A=A/A
B=A+A
Operator precedence grammar
PS C:\Users\adity\Documents\compiler>

PRACTICAL 10

AIM: Write a program to implement Shift Reduce parsing for a String.

Program:

```
#include <stdio.h>
#include <string.h>
struct ProductionRule
{
   char left[10];
   char right[10];
};
int main()
{
   char input[20], stack[50], temp[50], ch[2], *token1, *token2, *substring;
   int i, j, stack_length, substring_length, stack_top, rule_count = 0;
   struct ProductionRule rules[10];
   stack[0] = '\0';
   // User input for the number of production rules
printf("\nEnter the number of production rules: ");
scanf("%d", &rule_count);
   // User input for each production rule in the form 'left->right'
printf("\nEnter the production rules (in the form 'left->right'): \n");
   for (i = 0; i<rule_count; i++)
   {
scanf("%s", temp);
      token1 = strtok(temp, "->");
      token2 = strtok(NULL, "->");
strcpy(rules[i].left, token1);
strcpy(rules[i].right, token2);
   }
   // User input for the input string
printf("\nEnter the input string: ");
scanf("%s", input);
i = 0;
   while (1)
   {
      // If there are more characters in the input string, add the next character to the stack
      if (i<strlen(input))
      {
ch[0] = input[i];
ch[1] = '\0';
i++;
strcat(stack, ch);
printf("%s\t", stack);
         for (int k = i; k <strlen(input); k++)
         {
printf("%c", input[k]);
         }
```

```
printf("\tShift %s\n", ch);
        }
        // Iterate through the production rules
        for (j = 0; j <rule_count; j++)
        {
            // Check if the right-hand side of the production rule matches a substring in the
stack
            substring = strstr(stack, rules[j].right);
            if (substring != NULL)
            {
                // Replace the matched substring with the left-hand side of the production rule
stack_length = strlen(stack);
substring_length = strlen(substring);
stack_top = stack_length - substring_length;
                stack[stack_top] = '\0';
strcat(stack, rules[j].left);
printf("%s\t", stack);
                for (int k = i; k <strlen(input); k++)
                {
printf("%c", input[k]);
                }
printf("\tReduce %s->%s\n", rules[j].left, rules[j].right);
                j = -1; // Restart the loop to ensure immediate reduction of the newly derived
production rule
            }
        }
        // Check if the stack contains only the start symbol and if the entire input string has
been processed
        if (strcmp(stack, rules[0].left) == 0 &&i == strlen(input))
        {
printf("\nAccepted");
            break;
        }
        // Check if the entire input string has been processed but the stack doesn't match the
start symbol
        if (i == strlen(input))
        {
printf("\nNot Accepted");
            break;
        }
    }
    return 0;
}
```

OUTPUT:

```
Enter the number of production rules: 4

Enter the production rules (in the form 'left->right'):
E->E+E
E->E*E
E->(E)
E->i

Enter the input string: i+i*i
i       +i*i    Shift i
E       +i*i    Reduce E->i
E+      i*i     Shift +
E+i     *i      Shift i
E+E     *i      Reduce E->i
E       *i      Reduce E->E+E
E*      i       Shift *
E*i             Shift i
E*E             Reduce E->i
E               Reduce E->E*E

Accepted
```