Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

COMPILER DESIGN LABORATORY (303105350)

SEMESTER VI
Computer Science and Engineering Department

Laboratory Manual Session:2025-26

COMPILER DESIGN PRACTICAL BOOK

COMPUTER SCIENCE &ENGINEERING DEPARTMENT

PREFACE

It gives us immense pleasure to present the first edition of the **COMPILER DESIGN LABORATORY** Practical Book for the B.Tech . 6**TH** **semester** students for **PARUL UNIVERSITY**.

The **CD** theory and laboratory courses at PARUL UNIVERSITY, WAGHODIA, VADODARA are designed in such a way that students develop the basic understanding of the subject in the theory classes and then try their hands on the experiments to realize the various implementations of problems learnt during the theoretical sessions. The main objective of the **COMPILER DESIGN** laboratory course is: Learning **COMPILER DESIGN** through Experimentations. All the experiments are designed to illustrate various problems in different areas of **COMPILER DESIGN** and also to expose the students to various uses.

The objective of this **COMPILER DESIGN** Practical Book is to provide a comprehensive source for all the experiments included in the **COMPILER DESIGN** laboratory course. It explains all the aspects related to every experiment such as: basic underlying concept and how to analyze a problem. It also gives sufficient information on how to interpret and discuss the obtained results.

We acknowledge the authors and publishers of all the books which we have consulted while developing this Practical book. Hopefully this **COMPILER DESIGN** Practical Book will serve the purpose for which it has been developed.

INSTRUCTIONS TO STUDENTS

1. The main objective of the **COMPILER DESIGN** laboratory is: Learning through the Experimentation. All the experiments are designed to illustrate various problems in different areas of **COMPILER DESIGN** and also to expose the students to various problems and their uses.
2. Be prompt in arriving to the laboratory and always come well prepared for the practical.
3. Every student should have his/her individual copy of the **COMPILER DESIGN** Practical Book.
4. Every student have to prepare the notebooks specifically reserved for the **COMPILER DESIGN practical** work:" **COMPILER DESIGN** Practical Book"
5. Every student has to necessarily bring his/her **COMPILER DESIGN** Practical Book, **COMPILER DESIGN Practical** Class Notebook and **COMPILER DESIGN** Practical Final Notebook.
6. Finally find the output of the experiments along with the problem and note results in the **COMPILER DESIGN** Practical Notebook.
7. The grades for the **COMPILER DESIGN** practical course work will be awarded based on our performance in the laboratory, regularity, recording of experiments in the **COMPILER DESIGN Practical** Final Notebook, lab quiz, regular viva-voce and end-term examination.

Faculty of Engineering & Technology
Subject Name: Compiler Design Laboratory

Subject Code: 303105350

B.Tech. CSE Year: 3rd / Semester: 6th

CERTIFICATE

Mr. _____with Enrollment No. _____ has successfully completed his/her laboratory experiments ***Compiler Design Laboratory (303105350)*** *from the department of* ***Computer Science and Engineering*** *during the academic year* ***2024-2025.***

Date of Submission …… **Staff In charge ……**

Head of Department……

INDEX

Sr. No	Experiment Title	Page No		Start Date	Completion Date	Sign	Marks/ 10
		From	To				
1	Program to implement Lexical Analyzer.						

2	Program to count digits, vowels and symbols in C.						
3	Program to check validation of User Name and Password in C.						
4	Program to implement Predictive Parsing LL (1) in C.						
5	Program to implement Recursive Descent Parsing in C.						
6	Program to implement Operator Precedence Parsing in C.						
7	Program to implement LALR Parsing in C.						
8	To Study about Lexical Analyzer Generator (LEX) and Flex (Fast Lexical Analyzer)						
9	Implement following programs using Lex. a. Create a Lexer to take input from text file and count no of characters, no. of lines & no. of words. b. Write a Lex program to count number of vowels and consonants in a given input string.						
10	Implement following programs using Lex. a. Write a Lex program to print out all numbers from the given file. b. Write a Lex program to printout all HTML tags in file. c. Write a Lex program which adds line numbers to the given file and display the same onto the standard output.						

Experiment No: 1

Aim: - Program to implement Lexical Analyzer.

Code:

```
#include <ctype.h>
```

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LENGTH 100 bool
isDelimiter(char chr)
{
    return (chr == ' ' || chr == '+' || chr == '-'
          || chr == '*' || chr == '/' || chr == ','
          || chr == ';' || chr == '%' || chr == '>'
          || chr == '<' || chr == '=' || chr == '('
          || chr == ')' || chr == '[' || chr == ']'
          || chr == '{' || chr == '}');
}
bool isOperator(char chr)
{
    return (chr == '+' || chr == '-' || chr == '*'
          || chr == '/' || chr == '>' || chr == '<'
          || chr == '=');
}
bool isValidIdentifier(char* str)
{
   return (str[0] != '0' && str[0] != '1' && str[0] != '2'
         && str[0] != '3' && str[0] != '4'
         && str[0] != '5' && str[0] != '6'
         && str[0] != '7' && str[0] != '8'
         && str[0] != '9' && !isDelimiter(str[0]));
}
bool isKeyword(char* str)
{
   const char* keywords[]
      = { "auto", "break", "case", "char", "const",
          "continue", "default", "do",
          "double", "else", "enum", "extern",
          "float", "for", "goto", "if",
          "int", "long", "register", "return",
          "short", "signed", "sizeof", "static",
          "struct", "switch", "typedef", "union",
          "unsigned", "void", "volatile", "while" }; for
   (int i = 0;
       i < sizeof(keywords) / sizeof(keywords[0]); i++) {
      if (strcmp(str, keywords[i]) == 0) {
         return true;
      }
```

```
    }
    return false;
}
bool isInteger(char* str)
{
    if (str == NULL || *str == '\0') {
        return false;
    }
    int    i    =    0;    while
    (isdigit(str[i])) {
        i++;
    }
    return str[i] == '\0';
}
char* getSubstring(char* str, int start, int end)
{
    int   length   =   strlen(str);   int
    subLength  =  end  -  start  +  1;
    char* subStr
        = (char*)malloc((subLength + 1) * sizeof(char));
    strncpy(subStr, str + start, subLength); subStr[subLength]
    = '\0';
    return subStr;
}
int lexicalAnalyzer(char* input)
{
    int left = 0, right = 0; int len =
    strlen(input); while (right <= len &&
    left <= right) {
        if (!isDelimiter(input[right]))
            right++;
        if (isDelimiter(input[right]) && left == right) { if
            (isOperator(input[right]))
                printf("Token: Operator, Value: %c\n", input[right]);
            right++;
            left = right;
        }
        else if (isDelimiter(input[right]) && left != right
                || (right == len && left != right)) { char*
            subStr
                = getSubstring(input, left, right - 1);
            if    (isKeyword(subStr))    printf("Token:
                Keyword, Value: %s\n", subStr);
            else if (isInteger(subStr)) printf("Token:
                Integer, Value: %s\n", subStr);
```

```
        else if (isValidIdentifier(subStr)
              &&  !isDelimiter(input[right  -  1]))
           printf("Token: Identifier, Value: %s\n",
           subStr);
        else if (!isValidIdentifier(subStr)
              && !isDelimiter(input[right - 1])) printf("Token:
           Unidentified, Value: %s\n", subStr);
        left = right;
      }
    }
    return 0;
}
int main()
{
   char lex_input[MAX_LENGTH] = "int a = b + c"; printf("For
   Expression \"%s\":\n", lex_input);
   lexicalAnalyzer(lex_input);
   printf(" \n"); return
   (0); }
```

Output:

```
For Expression "int a = b + c":
Token: Keyword, Value: int
Token: Identifier, Value: a
Token: Operator, Value: =
Token: Identifier, Value: b
Token: Operator, Value: +
Token: Identifier, Value: c
```

Conclusion:

In this experiment, we successfully implemented a lexical analyzer in C to tokenize a given input string into meaningful components such as keywords, identifiers, numbers, and special characters.

Experiment No: 2

Aim: - Program to count digits, vowels and symbols in C.

Code:

```
#include <stdio.h> #include
<ctype.h>
void countCharacters(const char* input, int* digitCount, int* vowelCount, int*
symbolCount) {
```

```
    *digitCount = 0;
    *vowelCount = 0;
    *symbolCount = 0;
   for (int i = 0; input[i] != '\0'; i++) { char
      ch = input[i];
      if (isdigit(ch)) {
         (*digitCount)++;
      } else if (isalpha(ch)) { ch = tolower(ch); // Convert to lowercase
         for case-insensitivity
         if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            (*vowelCount)++;
         }
      } else if (!isspace(ch)) { // Ignore spaces
         (*symbolCount)++;
      }
   }
}
int main() {
   char input[200];
   int digitCount, vowelCount, symbolCount;
   printf("Enter a string: ");
   fgets(input, sizeof(input), stdin); // Read input string
   countCharacters(input, &digitCount, &vowelCount, &symbolCount);
   printf("Digits: %d\n", digitCount); printf("Vowels: %d\n",
   vowelCount); printf("Symbols: %d\n", symbolCount);
   return 0;
}
```

Output:

```
Enter a string: hello world @466
Digits: 3
Vowels: 3
Symbols: 1
```

Conclusion: In this experiment, we successfully implemented the program effectively counts digits, vowels, and symbols in a given string.

Experiment No: 3

Aim: - Program to check validation of User Name and Password in C.

Code :

```
#include <stdio.h>
#include <string.h>
#define USERNAME "sanjit"
#define PASSWORD "sanjit123"

int main() {
   char inputUsername[50], inputPassword[50];
   printf("Enter   username:   ");
   scanf("%s", inputUsername);
   printf("Enter   password:   ");
   scanf("%s", inputPassword);
   if (strcmp(inputUsername, USERNAME) == 0 && strcmp(inputPassword,
PASSWORD) == 0) { printf("Access
      Granted.\n");
   } else {
      printf("Invalid Username or Password.\n");
   }
   return 0;
}
```

Output :

```
Enter username: sanjit
Enter password: sanjit123
Access Granted.

--------------------------------
```

```
Enter username: abc
Enter password: 1234
Invalid Username or Password.

--------------------------------
```

Conclusion:

This C program validates a username and password by comparing user input with predefined credentials using strcmp. If the input matches, it grants access; otherwise, it denies access with an error message.

EXPERIMENT NO. 4 Aim:

Program to implement Predictive Parsing LL (1) in C.

Program:

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

void followfirst(char, int, int);
void  findfirst(char,  int,  int);
void follow(char);
void display_table();

int count, n = 0;
char calc_first[10][100], calc_follow[10][100]; int
m = 0;
char production[10][10], first[10]; char
f[10];
int k, e;
char done[10]; int
sid = 0;
char ter[10];

int main() { int
   i, choice;
   char ch;

   // Input grammar
   printf("How many productions ? : "); scanf("%d",
   &count);
   printf("\nEnter %d productions in form A=B where A and B are grammar
symbols:\n\n", count);
   for (i = 0; i < count; i++) {
      scanf("%s%c", production[i], &ch);
   }

   // Initialize tables for FIRST and FOLLOW sets
   for (k = 0; k < 10; k++) { for
      (int j = 0; j < 100; j++) {
      calc_first[k][j] = '!';
         calc_follow[k][j] = '!';
      }
```

```
}
// Compute FIRST sets int
ptr = -1; for (k = 0; k <
count; k++) {
   char c = production[k][0]; int xxx =
   0; for (int kay = 0; kay <= ptr;
   kay++) {
      if (c == done[kay]) {
         xxx = 1;
         break;
      }
   }
   if (xxx == 1) continue;
   findfirst(c, 0, 0); ptr += 1;
   done[ptr] = c;
   printf("\nFirst(%c) = { ", c);
   for (i = 0; i < n; i++) {
      printf("%c ", first[i]);
   }
   printf("}\n");
}

// Compute FOLLOW sets
ptr = -1; for (k = 0; k <
count; k++) {
   char ck = production[k][0];
   int xxx = 0;
   for (int kay = 0; kay <= ptr; kay++) {
      if (ck == done[kay]) {
         xxx = 1;
         break;
      }
   }
   if (xxx == 1) continue;
   follow(ck); ptr += 1;
   done[ptr] = ck;
   printf("Follow(%c) = { ",
   ck); for (i = 0; i < m; i++) {
      printf("%c ", f[i]);
   }
   printf("}\n");
}
```

```
                    // Calculate the LL(1) parsing table for
                    (k = 0; k < 10; k++) {
                      ter[k] = '!';
                    }

                    sid = 0;
                    for (k = 0; k < count; k++) { for (int kay = 0; kay < count; kay++) { if
                      (!isupper(production[k][kay]) && production[k][kay] != '#' &&
                  production[k][kay] != '=') {
                            int vp = 0;
                            for (int ap = 0; ap < sid; ap++) { if
                              (production[k][kay] == ter[ap]) {
                              vp = 1;
                                 break;
                              }
                            }
                            if (vp == 0) { ter[sid] =
                              production[k][kay]; sid++;
                            }
                         }
                      }
                    }
                    ter[sid] = '$';
                    sid++;
display_table();

                    // Parse input string char
                    input[100];
                    printf("\nEnter the desired input string: "); scanf("%s",
                    input);
                    input[strlen(input)] = '$'; // End of string symbol


                  printf("\n\t\t\t\t\t==========================================================
                   ====================\n");
                    printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction\n");

                   printf("\t\t\t\t\t==========================================================
                   ==================\n");

                    int i_ptr = 0, s_ptr = 1;
                    char stack[100]; stack[0]
                    = '$';
                    stack[1] = production[0][0];
```

```
                    while (s_ptr != -1) {
                      printf("\t\t\t\t\t\t");
                      for (int vamp = 0; vamp <= s_ptr; vamp++) {
                        printf("%c", stack[vamp]);
                      }
                      printf("\t\t\t");

                      int vamp = i_ptr; while
                      (input[vamp] != '\0') {
                        printf("%c", input[vamp]);
                        vamp++;
                      }
printf("\t\t\t");

                      char her = input[i_ptr]; char
                      him = stack[s_ptr];
                      s_ptr--;

                      if (!isupper(him)) {
                        if (her == him) {
                          i_ptr++;
                          printf("POP ACTION\n");
                        } else { printf("\nString Not Accepted by LL(1)
                          Parser!!\n"); exit(0);
                        }
                      } else { for (i = 0; i < sid;
                        i++) { if (ter[i] == her)
                              break;
                        }
for (int j = 0; j < sid; j++) {
                          if (him == table[j][0]) { if (table[j][i
                            + 1] == '#') { printf("%c=#\n",
                            table[j][0]); stack[s_ptr++] =
                            '#';
                            } else if (table[j][i + 1] != '!') { int
                              mum = (int)(table[j][i + 1]);
                              mum -= 65;
                              printf("%s\n", production[mum]);
                            } else { printf("\nString is Not Accepted by LL(1) Parser
                              !!\n"); exit(0);
                            }
                          }
                        }
                      }
```

```
    }

printf("\n\t\t\t==========================================================
=======================\n");
    if (input[i_ptr] == 'x') {
        printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED!!\n");
    } else {
        printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED!!\n");
    }

    return 0;
}

// Function to display the LL(1) parsing table void display_table() {
printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammar:\n");

printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\
n");

printf("\t\t\t\t\t==========================================================\
n");

    // Table printing logic (display format and table structure)
    printf("\t\t\t|\t"); for (int i = 0; i < sid; i++) { printf("%c\t",
    ter[i]);
    }

printf("\n\t\t\t==========================================================\n
"); for (int i = 0; i < count; i++) {
    printf("\t\t\t|\t%c\t", production[i][0]);
        for (int j = 0; j < sid; j++) { printf("%c\t",
            table[i][j]);
        }

printf("\n\t\t\t==========================================================\n
");
    }
}

// Function to find the FIRST set for a given symbol
```

```
void findfirst(char c, int q1, int q2) {
   if (!isupper(c)) { first[n++] = c;
   }
   for (int j = 0; j < count; j++) { if
      (production[j][0] == c) { if
      (production[j][2] == '#') {
            if (production[q1][q2] == '\0') first[n++]
               = '#';
            else
               findfirst(production[q1][q2], q1, (q2 + 1));
         } else if (!isupper(production[j][2])) { first[n++]
            = production[j][2];
         } else { findfirst(production[j][2],
            j, 3);
         }
      }
   }
}

// Function to calculate the FOLLOW set for a given non-terminal
void follow(char c) { int i, j;
   if (production[0][0] == c) {
      f[m++] = '$';
   }
   for (i = 0; i < count; i++) { for (j = 2; j <
      strlen(production[i]); j++) { if (production[i][j]
      == c) { if (production[i][j + 1] != '\0') {
      followfirst(production[i][j + 1], i, (j + 2));
            }
            if (production[i][j + 1] == '\0' && c != production[i][0]) {
               follow(production[i][0]);
            } }}}
// Function to calculate FIRST set for a non-terminal followed by another
void followfirst(char c, int c1, int c2) { int k; if (!(isupper(c))) {
      f[m++] = c;
   } else { int i = 0, j = 1; for (i
      = 0; i < count; i++) { if
      (calc_first[i][0] == c)
            break;
      while (calc_first[i][j] != '!') {
         if (calc_first[i][j] != '#') {
         f[m++] = calc_first[i][j];


      }
```

```
} else { if (production[c1][c2] ==
   '\0')                              {
   follow(production[c1][0]);
   } else { followfirst(production[c1][c2], c1, c2
      + 1);
   }
}
j++;}}}
```

Output:

```
How many productions ? :3

Enter 3 productions in form A=B where A and B are grammar symbols :

A=aABc
B=aC
C=c

 First(A)= { a, }

 First(B)= { a, }

 First(C)= { c, }


 Follow(A) = { $, a,  }

 Follow(B) = { c,  }

 Follow(C) = { c,  }

                          The LL(1) Parsing Table for the above grammer :-

            ================================================================================
               |   a       c       $
            ================================================================================
             A  |   A=aABc

             B  |   B=aC

             C  |           C=c


Please enter the desired INPUT STRING = aabbcc

                  ================================================================================
                      Stack           Input           Action
                  ================================================================================
                      $A          aabbcc$         A=aABc
                      $cBAa           aabbcc$         POP ACTION
                      $cBA            abbcc$          A=aABc
                      $cBcBAa         abbcc$          POP ACTION
                      $cBcBA          bbcc$           B=aC
                      $cBcBCa         bbcc$
String Not Accepted by LL(1) Parser !!

=== Code Execution Successful ===
```

Conclusion:

In this experiment, The LL(1) Predictive Parsing experiment demonstrates a top-down parsing approach that relies on FIRST and FOLLOW sets to construct a parsing table. The parser efficiently processes an input string using a stack-based mechanism and a parsing table, verifying whether the string belongs to the given grammar.

EXPERIMENT NO. 5 Aim:

Program to implement recursive decent parsing in C.

Program:

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

static char c[10];
int input = 0;
void E(), EPRIME();

int main() {
   printf("Enter a String: ");
   scanf("%s",  c);  strcat(c,
   "$"); E();
   if (c[input] == '$') {
      printf("Valid String\n");
   }   else   {   printf("Invalid
      String\n");
   }
   return 0;
}

void E() { if (c[input]
   == 'i') {
      input++;
      EPRIME();
   }   else   {   printf("Invalid
      String\n");
      exit(0);
   }
}

void EPRIME() {
   if (c[input] == '+') { input++;
      if (c[input] == 'i') {
         input++;
         EPRIME();
      }   else   {   printf("Invalid
         String\n"); exit(0);
   } else if (c[input] == '$') { return;


      }
```

```
        } else { printf("Invalid
          String\n");
          exit(0);
        }
      }
```

Output:

```
Enter a String: $c
Valid String


=== Code Execution Successful ===
```

Conclusion:

In this experiment, The Recursive Descent Parser implemented in this experiment follows a top-down parsing approach using a set of mutually recursive functions to process the input string based on a given grammar. This parser is non-backtracking and is typically used for LL(1) grammars, which require only one token of lookahead.

EXPERIMENT NO. 6

Aim: Program to implement operator precedence parsing in C.

Program:

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

int main() {
   char stack[20], ip[20], opt[10][10], ter[10];
   int i, j, k, n, top = 0, row, col; int
   len;
for (i = 0; i < 10; i++) { stack[i] = '\0'; ip[i] = '\0';
   for (j = 0; j < 10; j++) {
   opt[i][j] = '\0';
     }
   }
```

```
printf("Enter the no. of terminals: ");
scanf("%d", &n);

printf("\nEnter the terminals: "); scanf("%s",
ter);

printf("\nEnter the table values:\n");
for (i = 0; i < n; i++) { for
   (j = 0; j < n; j++) {
      printf("Enter the value for %c %c: ", ter[i], ter[j]); scanf("
      %c", &opt[i][j]);
   }
}

printf("\nOPERATOR PRECEDENCE TABLE:\n"); for
(i = 0; i < n; i++) {
   printf("\t%c", ter[i]);
}
printf("\n ");

for (i = 0; i < n; i++) {
   printf("\n%c |", ter[i]); for
   (j = 0; j < n; j++) {
      printf("\t%c", opt[i][j]);
}
stack[top] = '$';

printf("\n\nEnter the input string (append with $): "); scanf("%s",
ip);
i = 0;

printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t", stack, ip);
len = strlen(ip);

while (i <= len) {
   for (k = 0; k < n; k++) {
      if (stack[top] == ter[k]) row = k;
      if (ip[i] == ter[k]) col = k;
   }

   if ((stack[top] == '$') && (ip[i] == '$')) {
   }
```

```
            printf("String is ACCEPTED"); break;
        } else if ((opt[row][col] == '<') || (opt[row][col] == '=')) { stack[++top]
            = opt[row][col];
            stack[++top] = ip[i];
            ip[i] = ' '; printf("Shift
            %c", ip[i]);
            i++;
        } else { if (opt[row][col] ==
            '>') {
                while (stack[top] != '<') { --top;
                }
                top = top - 1; printf("Reduce");
            } else { printf("\nString is not
                accepted"); break;
            }
        }

        printf("\n");
        printf("%s\t\t\t%s\t\t\t", stack, ip);
    }

    return 0;
}
```

Output:

```
Enter the no.of terminals:3

Enter the terminals:aba

Enter the table values:
Enter the value for a a:1
Enter the value for a b:2
Enter the value for a a:3
Enter the value for b a:4
Enter the value for b b:5
Enter the value for b a:6
Enter the value for a a:7
Enter the value for a b:8
Enter the value for a a:9

OPERATOR PRECEDENCE TABLE:
        a       b       a


a |     1       2       3
b |     4       5       6
a |     7       8       9


Enter the input string(append with $):$a

STACK                   INPUT STRING                    ACTION

$                       $a                      String is ACCEPTED

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

In this experiment, The Operator Precedence Parser implemented in this experiment demonstrates a bottom-up parsing technique that is used for a subset of context-free grammars. It relies on an operator precedence table to determine parsing actions like shift and reduce based on the precedence relations between terminals.

EXPERIMENT NO. 7 Aim:

Program to implement LALR parsing in C.

Program:

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

void push(char *, int *, char); char
stacktop(char *); void
isproduct(char, char); int ister(char);
int isnter(char); int isstate(char);
void error(); void isreduce(char,
char); char pop(char *, int *); void
printt(char *, int *, char [], int); void
rep(char [], int);

struct action {
   char row[6][5];
};
 const struct action A[12] = {
   {"sf", "emp", "emp", "se", "emp", "emp"},
   {"emp", "sg", "emp", "emp", "emp", "acc"},
   {"emp", "rc", "sh", "emp", "rc", "rc"},
   {"emp", "re", "re", "emp", "re", "re"},
   {"sf", "emp", "emp", "se", "emp", "emp"},
   {"emp", "rg", "rg", "emp", "rg", "rg"},
   {"sf", "emp", "emp", "se", "emp", "emp"},
   {"sf", "emp", "emp", "se", "emp", "emp"},
   {"emp", "sg", "emp", "emp", "sl", "emp"},
   {"emp", "rb", "sh", "emp", "rb", "rb"},
   {"emp", "rb", "rd", "emp", "rd", "rd"},
   {"emp", "rf", "rf", "emp", "rf", "rf"}
};

struct gotol {
   char r[3][4];
};
 const struct gotol G[12] = {
   {"b", "c", "d"},
   {"emp", "emp", "emp"},
   {"emp", "emp", "emp"},
   {"emp", "emp", "emp"},
   {"i", "c", "d"},
   {"emp", "emp", "emp"},
   {"emp", "j", "d"},
```

```
                    {"emp", "emp", "k"},
                    {"emp", "emp", "emp"},
                    {"emp", "emp", "emp"},
                };
char ter[6] = {'i', '+', '*', ')', '(', '$'}; char nter[3] = {'E',
                'T', 'F'};
                char states[12] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l'}; char
                stack[100];
                int top = -1;
                char temp[10];

                struct grammar { char
                  left;
                  char right[5];
                };
                const struct grammar rl[6] = {
                  {'E', "e+T"},
                  {'E', "T"},
                  {'T', "T*F"},
                  {'T', "F"},
                  {'F', "(E)"},
                  {'F', "i"},
                };

                void main() {
                  char inp[80], x, p, dl[80], y, bl = 'a';
                  int i = 0, j, k, l, n, m, c, len;
                  printf("Enter the input: ");
                  scanf("%s", inp);

                  len = strlen(inp); inp[len]
                  = '$';
                  inp[len + 1] = '\0';

                  push(stack,                &top,                bl);
                  printf("\nStack\t\t\tInput\n");
                  printt(stack, &top, inp, i);

                  do {
                    x = inp[i]; p =
                    stacktop(stack);
                    isproduct(x, p);

                    if (strcmp(temp, "emp") == 0)
```

```
            error();

         if (strcmp(temp, "acc") == 0)
            break;
         else {
            if (temp[0] == 's') {
               push(stack, &top, inp[i]);
               push(stack, &top, temp[1]); i++;
            } else { if (temp[0] ==
               'r') {
                  j = isstate(temp[1]);
                  strcpy(temp, rl[j - 2].right);
                  dl[0] = rl[j - 2].left; dl[1] =
                  '\0'; n = strlen(temp);

                  for (k = 0; k < 2 * n; k++)
                     pop(stack, &top);

                  for (m = 0; dl[m] != '\0'; m++)
                     push(stack, &top, dl[m]);

                  l = top; y =
                  stack[l - 1];
                  isreduce(y, dl[0]);

                  for (m = 0; temp[m] != '\0'; m++)
                     push(stack, &top, temp[m]);
               }
            }
         }
         printt(stack, &top, inp, i);
      } while (inp[i] != '\0');

      if (strcmp(temp, "acc") == 0)
         printf("\nAccept the input");
      else
         printf("\nDo not accept the input");
   }

   void push(char *s, int *sp, char item) {
      if (*sp == 100)
         printf("Stack is full\n");
      else { *sp = *sp
         + 1;
```

```
        s[*sp] = item;
    }
}

char stacktop(char *s) {
    return s[top];
}

void isproduct(char x, char p) {
    int k, l; k = ister(x); l =
    isstate(p);
    strcpy(temp, A[l - 1].row[k - 1]);
}

int ister(char x) { for (int
    i = 0; i < 6; i++)
        if (x == ter[i])
            return i + 1;
    return 0;
}

int isnter(char x) { for
    (int i = 0; i < 3; i++)
        if (x == nter[i])
            return i + 1;
    return 0;
}

int isstate(char p) { for (int
    i = 0; i < 12; i++)
        if (p == states[i])
            return i + 1;
    return 0;
}

void error() { printf("Error in the
    input\n");
    exit(0);
}

void isreduce(char x, char p) { int
    k, l;
    k = isstate(x); l
    = isnter(p);
```

```
    strcpy(temp, G[k - 1].r[l - 1]);
}

char pop(char *s, int *sp) {
    char item; if
    (*sp == -1)
        printf("Stack is empty\n");
    else {
        item = s[*sp];
        *sp = *sp - 1;
    }
    return item;
}

void printt(char *t, int *p, char inp[], int i) {
    int r; printf("\n"); for
    (r = 0; r <= *p; r++)
        rep(t, r);

    printf("\t\t\t"); for (r = i;
    inp[r] != '\0'; r++)
        printf("%c", inp[r]);
}

void rep(char t[], int r) {
    char c = t[r]; switch
    (c) {
        case 'a': printf("0"); break;
        case 'b': printf("1"); break;
        case 'c': printf("2"); break;
        case 'd': printf("3"); break;
        case 'e': printf("4"); break;
        case 'f': printf("5"); break;
        case 'g': printf("6"); break;
        case 'h': printf("7"); break;
        case 'm': printf("8"); break;
        case 'j': printf("9"); break; case
        'k': printf("10"); break; case 'l':
        printf("11"); break;
        default: printf("%c", t[r]); break;
    }
}
```

Output:

```
Enter the input :i+i+i

stack                    input
0                        i+i+i$
0i5                      +i+i$
0F3                      +i+i$
0T2                      +i+i$
0E1                      +i+i$
0E1+6                    i+i$
0E1+6i5                  +i$
0E1+6F3                  +i$
0E1+6T9                  +i$
0E1                      +i$
0E1+6                    i$
0E1+6i5                  $
0E1+6F3                  $
0E1+6T9                  $
0E1                      $
accept the input

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

In this experiment, The LALR (Look-Ahead LR) Parsing experiment demonstrates the implementation of an LALR(1) parser, which is an optimized version of LR(1) parsing. LALR parsers are widely used in compiler design as they reduce the number of states compared to canonical LR(1) parsing while maintaining similar parsing power.

EXPERIMENT NO. 8 Aim:

Program to implement LALR parsing in C.

(A) LEX:

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

Syntax:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines } **Sample lex program:**

```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}
%%
([a-zA-Z0-9])* {i++;} "\n"
{printf("%d\n", i);
i = 0;}
%%
int yywrap(void){} int
main()
{
yylex();
return 0;
}
```

(B) FLEX:

- FLEX is a tool/computer program for generating lexical analyzers. Flex and Bison both are more flexible than Lex and Yacc and produces faster code. Bison produces parser from the input file provided by the user. The function yylex() is automatically generated by the flex when it is provided with a .l file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.
- The function yylex() is the main flex function which runs the Rule Section and extension (.l) is the extension used to save the programs.

Program Structure:

In the input file, there are 3 sections:

1. Definition Section:

The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{ %}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c

2. Rules Section:

The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in "%% %%".

3. User Code Section:

This section contain C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

How to run the program:

To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.

Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with.
Step 2: gcc lex.yy.c
Step 3: ./a.out

Experiment No: 9 Aim:

- Implement following programs using Lex.

a) Create a Lexer to take input from text file and count no of characters, no. of lines & no. of words.

Code :

```
%{
    #include<stdio.h>
    int wordc = 0, charc = 0, spacec = 0, linec = 0;
%}
%%
[\n]       { linec++; charc++; }
```

```
[ \t]       { spacec++; charc++; }
[^ \t\n]+    { wordc++; charc += yyleng; }
%%
int main() { printf("Enter the input (press Ctrl+D to
    end):\n"); yylex();
    printf("Number of words: %d\n", wordc);
    printf("Number of characters: %d\n", charc);
    printf("Number of spaces: %d\n", spacec);
    printf("Number of lines: %d\n", linec); return
    0;
}
int yywrap() { return
    1;
}
```

Output :

Conclusion:

The implemented Lex program successfully counts the number of words, characters, spaces, and lines in a given input text. The program works by scanning through the input text and applying regular expressions to detect different character types, including newlines, spaces, and words.

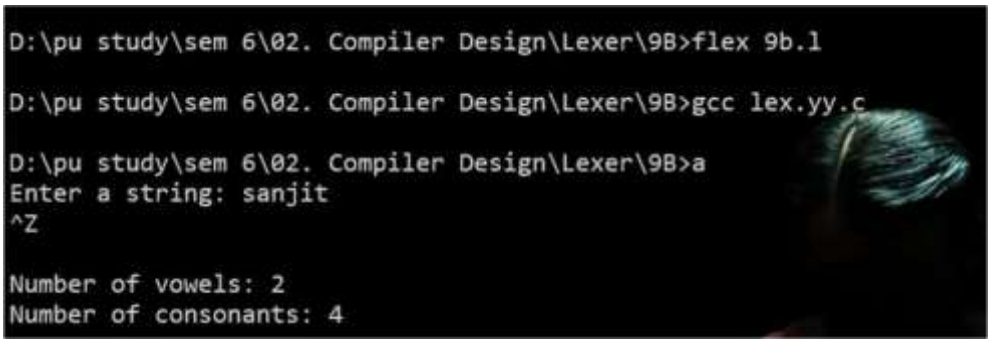
b) . Write a Lex program to count number of vowels and consonants in a given input string.

Code :

```
%{

#include <stdio.h>
int vowels = 0, consonants = 0; // Variables to store counts
%}
%%
[aeiouAEIOU]    { vowels++; }      // Increment vowels count for vowels
[a-zA-Z]        { consonants++; } // Increment consonants count for other letters
.|\n            ;                 // Ignore other characters
%%
int main() { printf("Enter a string: "); yylex(); // Start
   scanning input printf("\nNumber of vowels: %d\n",
```

```
    vowels); printf("Number of consonants: %d\n",
    consonants); return 0;
}
int yywrap() {
    return 1;
}
```

Output :



Conclusion:

The Lex program effectively counts the number of vowels and consonants in a given input string. It uses regular expressions to identify vowel characters (aeiouAEIOU) and consonants (a-zA-Z excluding vowels). The program then outputs the total count of vowels and consonants, ignoring other characters. This solution provides a simple and efficient way to analyze and categorize characters in a string based on their type.

EXPERIMENT NO. 10-A

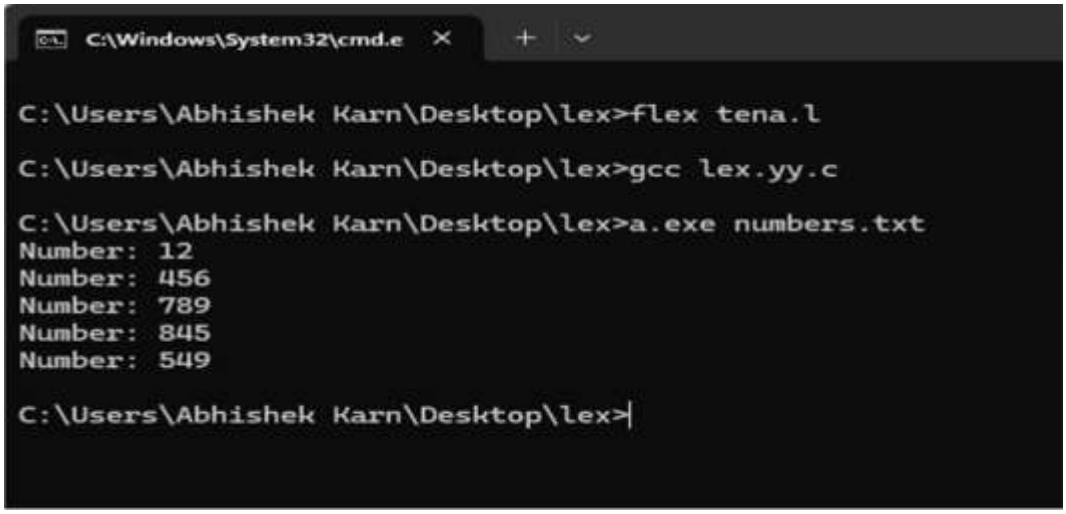
Aim: Write a Lex program to print out all numbers from the given file.

Program:

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

%%

[0-9]+  { printf("Number: %s\n", yytext); }
.|\n    { /* Ignore everything else */ }

%%

int yywrap() { return
        1;
}
```

```
int main(int argc, char *argv[]) { if
   (argc > 1) {
      FILE *file = fopen(argv[1], "r");
      if (!file) {
         fprintf(stderr, "Error: Cannot open file %s\n", argv[1]); return
         1;
      }
      yyin = file; // Set yyin to the file stream
   } else { yyin = stdin; // If no file argument is provided, use
      stdin
   }

   yylex(); // Start lexing the input return
   0;
}
```

Output:



Conclusion:

In this experiment, we successfully implemented a Lex program to extract and print all the numbers from a given file. The program utilized Lex's pattern-matching capabilities to identify sequences of digits in the input file and print them to the output.

EXPERIMENT NO. 10-B

Aim: Write a Lex program to printout all HTML tags in file.

Program:

```
%{
int tags = 0; // Initialize tags count
%}

%%

"<"[^>]*">" { tags++; printf("%s\n", yytext); } // Count and print each tag .|\n {
} // Ignore everything else

%%

int yywrap(void) { return 1; // End of
input handling }

int main(void) { FILE
   *f;
   char file[100]; // Increase the buffer size for the filename input
   printf("Enter File Name: "); scanf("%s", file);

   f = fopen(file, "r"); // Open the file in read mode if (!f)
   { // Check if the file was successfully opened
   fprintf(stderr, "Error: Cannot open file %s\n", file);
   return 1;
   }

   yyin = f; // Set yyin to the file pointer yylex();
   // Start lexing the file
printf("\nNumber of HTML tags: %d\n", tags); // Print the number of HTML tags

   fclose(f); // Close the file after lexing is done return
   0;
}
```

Output:

Conclusion:

In this experiment, we successfully implemented a Lex program to extract and print all HTML tags from a given file. The program utilized Lex's powerful pattern-matching capabilities to identify and process HTML tags accurately. By defining rules to match HTML tag patterns, the program was able to distinguish tags from the rest of the content and output them effectively.

EXPERIMENT NO. 10-C

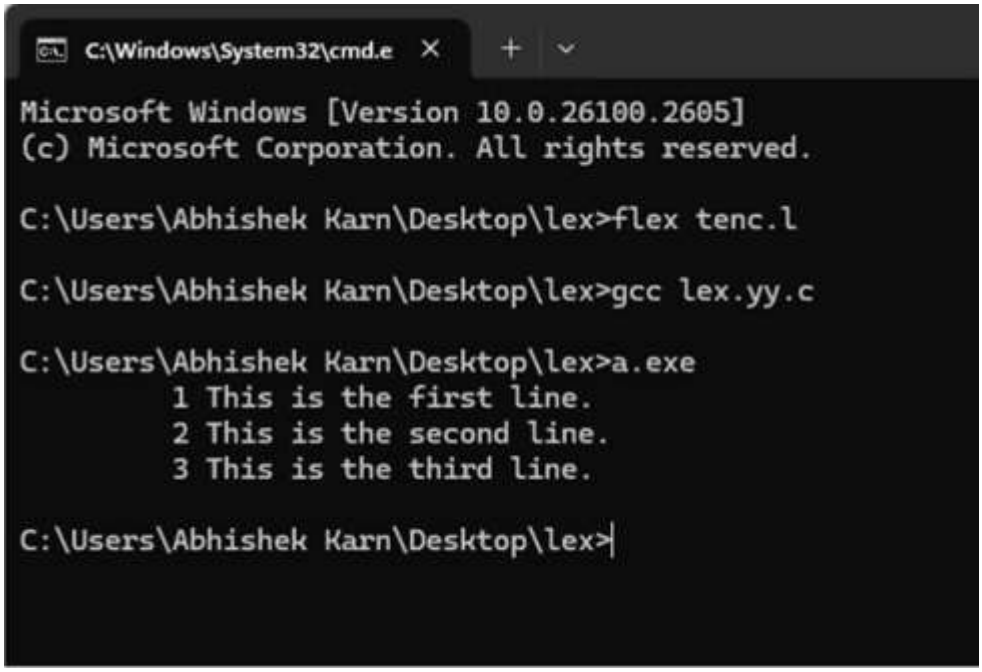
Aim: Write a Lex program which adds line numbers to the given file and display the same onto the standard output.

Program:

```
%{
int line_number = 1; // initializing line number to 1 %}

%%
```

```
.|\n { printf("%10d %s", line_number++, yytext); } // Match each character, including line
breaks
%%

int yywrap() { return 1; // Return 1 to
   indicate EOF
}

int main(int argc, char *argv[]) { extern FILE
   *yyin; // Declare external yyin

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

   // Open the input file passed as argument
   yyin = fopen(argv[1], "r");
   if (!yyin) { fprintf(stderr, "Error: Cannot open file %s\n",
      argv[1]); return 1;
   }
yylex(); // The function that starts the analysis

   fclose(yyin); // Close the file after lexing is done return
   0;
}
```

Output:



Conclusion:

In this experiment, we successfully developed a Lex program that reads an input file, adds line numbers to each line, and displays the numbered lines on the standard output. The program effectively utilized Lex's pattern-matching capabilities to process each line of the file, incrementing a line counter to ensure accurate numbering.