COMPILER DESIGN LAB

At

BABA BANDA SINGH BAHADUR ENGINEERING COLLEGE

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT

FOR THE

BACHELOR OF TECHNOLOGY

(Computer Science & Engineering)

SUBMITTED TO:-Prof.
ANUPMA SHARMA

SUBMITTED BY:-
Name:- PRINCE KUMAR
Roll No:- 2001308
Group:- 6C23
Branch:- CSE

BABA BANDA SINGH BAHADUR ENGINEERING COLLEGE

FATEHGARH SAHIB

PRACTICAL 1.

AIM :- WAP TO COUNT OF BLANK SPACE , NO. OF VOWELS , NO. OF NEW LINES AND NO. OF CONSONANTS IN A PARAGRAPH.

```
#include <stdio.h>

int main() {
   char paragraph[1000];
   int num_spaces = 0, num_vowels = 0, num_newlines = 0, num_consonants = 0;
   char vowels[] = "aeiouAEIOU";

   printf("Enter a paragraph:\n");
   fgets(paragraph, 1000, stdin);

   for (int i = 0; paragraph[i] != '\0'; i++) {
      if (paragraph[i] == ' ') {
         num_spaces++;
      } else if (strchr(vowels, paragraph[i]) != NULL) {
         num_vowels++;
      } else if (paragraph[i] == '\n') {
         num_newlines++;
      } else if (isalpha(paragraph[i])) {
         num_consonants++;
      }
   }

   printf("Number of spaces: %d\n", num_spaces);
   printf("Number of vowels: %d\n", num_vowels);
   printf("Number of new lines: %d\n", num_newlines);
   printf("Number of consonants: %d\n", num_consonants);

   return 0;
}
```

OUTPUT.

```
Enter a paragraph:
I am a new student in turbo C++
Number of spaces: 7
Number of vowels: 9
Number of new lines: 1
Number of consonants: 13
```

PRACTICAL 2.

AIM :- WAP TO FIND OUT WETHER A GIVEN LINE OF CODE IS COMMENT OR NOT.

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

int main() {
   char line[100];

   printf("Enter a line of code: ");
   fgets(line, sizeof(line), stdin);

   // Check if the line is a comment
   if (line[0] == '/' && line[1] == '/') {
      printf("The given line is a comment.\n");
   } else {
      printf("The given line is not a comment.\n");
   }

   return 0;
}
```

OUTPUT.

Enter a line of code: NEWCARBUY
The given line is not a comment**.**

PRACTICAL 3.

AIM:- WRITE A PROGRAM TO RECOGNIZE THE STRINGS UNDER a* / a*b+ / abb.

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

int main() {
   char str[100];

   printf("Enter a string: ");
   fgets(str, sizeof(str), stdin);

   // Recognize the string under a*
   if (str[0] == 'a' || str[0] == '\n') {
      printf("The string '%s' is under a*.\n", str);
   }

   // Recognize the string under a*b+
   if (str[0] == 'a' && (str[1] == 'b' || str[1] == '\n')) {
      int i;
      for (i = 2; i < strlen(str) - 1; i++) {
         if (str[i] != 'b') {
            printf("The string '%s' is not under a*b+.\n", str);
            return 0;
         }
      }
      printf("The string '%s' is under a*b+.\n", str);
   }

   // Recognize the string under abb
   if (strcmp(str, "abb\n") == 0) {
      printf("The string '%s' is under abb.\n", str);
   }

   return 0;
}
```

OUTPUT.

```
Enter a string: abbbbb
The string 'abbbbb
' is under a*.
The string 'abbbbb
' is under a*b+.
```

PRACTICAL 4.

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

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

int main() {
   char identifier[100];
   int i;

   printf("Enter an identifier: ");
   scanf("%s", identifier);

   // Check if the identifier starts with a letter or underscore
   if (!isalpha(identifier[0]) && identifier[0] != '_') {
      printf("The identifier '%s' is not valid.\n", identifier);
      return 0;
   }

   // Check if the rest of the identifier consists of letters, digits, or underscores
   for (i = 1; i < strlen(identifier); i++) {
      if (!isalnum(identifier[i]) && identifier[i] != '_') {
         printf("The identifier '%s' is not valid.\n", identifier);
         return 0;
      }
   }

   // If we made it this far, the identifier is valid
   printf("The identifier '%s' is valid.\n", identifier);

   return 0;
}
```

OUTPUT.

```
Enter an identifier: first
The identifier 'first' is valid.
```

PRACTICAL 5.

AIM- Write a C program to simulate lexical analyzer for validatin operators.

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

int main() {
   char operator[100];

   printf("Enter an operator: ");
   scanf("%s", operator);

   if (strcmp(operator, "+") == 0 || strcmp(operator, "-") == 0 || strcmp(operator, "*") == 0 ||
      strcmp(operator, "/") == 0 || strcmp(operator, "%") == 0 || strcmp(operator, "=") == 0 ||
      strcmp(operator, "==") == 0 || strcmp(operator, "!=") == 0 || strcmp(operator, ">") == 0 ||
      strcmp(operator, ">=") == 0 || strcmp(operator, "<") == 0 || strcmp(operator, "<=") == 0 ||
      strcmp(operator, "&&") == 0 || strcmp(operator, "||") == 0 || strcmp(operator, "!") == 0) {
      printf("The operator '%s' is valid.\n", operator);
   } else {
      printf("The operator '%s' is not valid.\n", operator);
   }

   return 0;
}
```

OUTPUT.

```
Enter an operator: +
The operator '+' is valid

Enter an operator: *
The operator '*' is valid

Enter an operator: -
The operator '-' is valid
```

PRACTICAL 6.

AIM:- Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

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

#define MAX_TOKEN_LENGTH 100

/* Symbolic constants for token types */
#define TOKEN_IDENTIFIER 1
#define TOKEN_KEYWORD 2
#define TOKEN_NUMBER 3
#define TOKEN_OPERATOR 4

/* Symbolic constants for keywords */
#define KEYWORD_IF 1
#define KEYWORD_ELSE 2
#define KEYWORD_WHILE 3
#define KEYWORD_FOR 4

/* Symbolic constants for operators */
#define OPERATOR_PLUS 1
#define OPERATOR_MINUS 2
#define OPERATOR_MULT 3
#define OPERATOR_DIV 4
%}

/* Regular expressions for identifiers, keywords, and numerical literals */

digit [0-9]
letter [a-zA-Z]
id {letter}({letter}|{digit})*
keyword if|else|while|for
number {digit}+(\.{digit}+)?([eE][+-]?{digit}+)?

/* Regular expressions for operators */
plus \+
minus -
mult \*
div /

%%

/* Rules for identifiers and keywords */
```

```
    {id} {

     yylval.strval = strdup(yytext);

if (!yylval.strval) {

        fprintf(stderr, "Out of memory\n");
        exit(1);
      }
      if (strcmp(yytext, "if") == 0) {
        return TOKEN_KEYWORD;
      } else if (strcmp(yytext, "else") == 0) {
        return TOKEN_KEYWORD;
      } else if (strcmp(yytext, "while") == 0) {
        return TOKEN_KEYWORD;
      } else if (strcmp(yytext, "for") == 0) {
        return TOKEN_KEYWORD;
      } else {
        return TOKEN_IDENTIFIER;
      }
    }

    /* Rules for numerical literals */
    {number} {
      yylval.numval = atof(yytext);
      return TOKEN_NUMBER;
    }

    /* Rules for operators */
    {plus} {
      return OPERATOR_PLUS;
    }

    {minus} {
      return OPERATOR_MINUS;
    }

    {mult} {
      return OPERATOR_MULT;
    }

    {div} {
      return OPERATOR_DIV;
    }

    /* Rule for whitespace */
    [ \t\n] {
      /* ignore whitespace */
    }
```

```
        /* Rule for invalid characters */

    . {
    fprintf(stderr, "Invalid character: %s\n", yytext);
          exit(1);
        %%
        /* Main program */
        int main(void) {
          int token_type;

          /* Loop through tokens in input */
          while (token_type = yylex()) {
            switch (token_type) {
              case TOKEN_IDENTIFIER:
                printf("Identifier: %s\n", yylval.strval);
                free(yylval.strval);
                break;
              case TOKEN_KEYWORD:
                printf("Keyword: %s\n", yylval.strval);
                free(yylval.strval);
                break;
              case TOKEN_NUMBER:
                printf("Number: %f\n", yylval.numval);
                break;
              case OPERATOR_PLUS:
                printf("Operator: +\n");
                break;
              case OPERATOR_MINUS:
                printf("Operator: -\n");
                break;
              case OPERATOR_MULT:
                printf("Operator: *\n");
                break;
              case OPERATOR_DIV:
                printf("Operator: /\n");
                break;
            }
          }

          return 0;
}
```

INPUT.

```
int x = 5;
if (x > 0) {
  printf("x is positive");
} else:
```

```
{
  printf("x is non-positive");}
```

OUTPUT.

```
Keyword: int
Identifier: x
Operator: =
Number: 5.000000
Keyword: if
Operator: (
Identifier: x
Operator: >
Number: 0.000000
Operator: )
Operator: {
Keyword: printf
Operator: (
"\"x is positive\""
Operator: )
Operator: ;
Operator: }
Keyword: else
Operator: {
Keyword: printf
Operator: (
"\"x is non-positive\""
Operator: )
Operator: ;
Operator: }
```

PRACTICAL 7.

AIM:- Write a program for implementing the functionalities of predictive parser for the mini language specified in Note 1.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char pror[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"};
char first[7][10]={"abcd","ab","cd","a@","@","c@","@"};
char follow[7][10]={"$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
numr(char c)
{
   switch(c)
{
case 'S': return 0; case 'A': return 1; case 'B': return 2; case 'C': return 3; case 'a': return 0;
case 'b': return 1; case 'c': return 2; case 'd': return 3; case '$': return 4;
}
 return(2);
}
void main()
{
   Int i, j, k;
Clrscr ();
for(i=0;i<5;i++) for(j=0;j<6;j++)
strcpy(table[i][j]," ");
printf("\nThe following is the predictive parsing table for the following grammar:\n");
for(i=0;i<7;i++)
printf("%s\n",prod[i]);
printf("\nPredictive parsing table is\n");
fflush(stdin);
```

```
for(i=0;i<7;i++)
{
k=strlen(first[i]); for(j=0;j<10;j++) if(first[i][j]!='@')
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
}
for(i=0;i<7;i++)
{
if(strlen(pror[i])==1)
{
if(pror[i][0]=='@')
{
k=strlen(follow[i]); for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
strcpy(table[3][0],"B");
strcpy(table[4][0],"C");
printf("\n\n");
for(i=0;i<5;i++) for(j=0;j<6;j++)
{
printf("%-10s",table[i][j]);
 if(j==5)
printf("\n\n"); } getch();
}
```

OUTPUT.

The following is the predictive parsing table for the following grammar: S->A
A->Bb A->Cd B->aB B->@ C->Cc C->@
Predictive parsing table is
a b c d $
S S->AS->AS->AS->A
A A->Bb A->BbA->Cd A->Cd
B B->aB B->@ B->@ B->@
C C->@C->@ C->@

PRACTICAL 8.

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

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
char s[20],stack[20];
void main()
{
char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," ","
","n","*fc"," a ","n","n","i"," "," ","(e)"," "," "};
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2; clrscr();
printf("\n Enter the input string: ");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
 i=1; j=0;
printf("\nStack Input\n");
printf("\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]==s[j])
{ i--; j++; }
switch(stack[i])
{
case 'e': str1=0; break;
case 'b': str1=1; break;
case 't': str1=2; break;
case 'c': str1=3; break;
```

```
case 'f': str1=4; break;
}
 switch(s[j])
{
case 'i': str2=0; break;
case '+': str2=1; break;
case '*': str2=2; break;
case '(': str2=3; break;
case ')': str2=4; break;
case '$': str2=5; break;
}
if(m[str1][str2][0]=='\0')
{ printf("\nERROR");
exit(0); }
else if(m[str1][str2][0]=='n') i--;
 else
if(m[str1][str2][0]=='i') stack[i]='i';
else {
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k]; i++;
}
i--; }
for(k=0;k<=i;k++)
printf(" %c",stack[k]);
printf(" "); for(k=j;k<=n;k++)
printf("%c",s[k]);
printf(" \n ");
}
printf("\n SUCCESS");
getch();
}
```

OUTPUT.


Enter the input string: i*i+i
Stack Input
$ b t i*i+i$
$ b c f i*i+i$
$ b c i i*i+i$
$ b c f * *i+i$
$ b c i i+i$
$ b +i$
$ b t + +i$
$ b c f i$
$ b c i i$
$ b $
SUCCESS_

PRACTICAL 9.

AIM:- Write a C program to implement LALR parsing.

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

#define MAX_SYMBOLS 100

/* Parse table */
int parse_table[MAX_SYMBOLS][MAX_SYMBOLS] = {
   /*   a     b    $  */
   /*  0 */ {  0,   1,  0 },
   /*  1 */ {  2,   3,  0 },
   /*  2 */ {  0,   0,  1 },
   /*  3 */ { -1,  -1,  1 }
};

/* Driver function for parsing */
void parse_input(char *input) {
   int stack[MAX_SYMBOLS];
   int top = 0;
   int i, j, k;
   int next_token;
   int action_type;
   int action_value;

   /* Initialize the stack with the start symbol */
   stack[top] = 0;

   /* Parse the input using the parse table */
   i = stack[top];
   j = *input - 'a' + 1;

   while (1) {
      action_type = parse_table[i][j];
      action_value = action_type & 0x7FFF;

      if (action_type == 0xFFFF) {
         printf("Error\n");
         exit(1);
      }

      if (action_type & 0x8000) {
         /* Reduce */
         for (k = 0; k < 2; k++) {
            top--;
```

```
                i = stack[top];
            }

            /* Push the nonterminal on top of the stack */
            stack[++top] = parse_table[i][action_value] & 0x7FFF;
        } else {
            /* Shift */
            stack[++top] = action_value;

            if (action_type == 0) {
                printf("Error\n");
                exit(1);
            }

            if (action_type == 1) {
                printf("Accepted\n");
                exit(0);
            }

            /* Move to the next token in the input */
            input++;
            j = *input - 'a' + 1;
        }

        i = stack[top - 1];
    }
}

int main() {
    char input[1024];
    printf("Enter input string: ");
    scanf("%s", input);
    parse_input(input);
    return 0;
}
```

OUTPUT.

```
Enter input string: aabb
Accepted.
```

PRACTICAL 10.

1. AIM:- Write a C program to implement operator precedence parsing.

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

#define MAX_EXPR_LENGTH 100

int precedence(char operator) {
   switch(operator) {
      case '+':
      case '-':
         return 1;
      case '*':
      case '/':
         return 2;
      case '^':
         return 3;
      default:
         return -1;
   }
}

void evaluate(char operator, int *operand_stack, int *top) {
   int result;
   int operand2 = operand_stack[*top];
   (*top)--;
   int operand1 = operand_stack[*top];
   (*top)--;

   switch(operator) {
      case '+':
         result = operand1 + operand2;
         break;
      case '-':
         result = operand1 - operand2;
         break;
      case '*':
         result = operand1 * operand2;
         break;
      case '/':
         result = operand1 / operand2;
         break;
      case '^':
         result = 1;
         for(int i = 1; i <= operand2; i++) {
            result *= operand1;
```

```
            }
            break;
        default:
            printf("Invalid operator!\n");
            exit(0);
    }

    operand_stack[++(*top)] = result;
}

void operatorPrecedenceParsing(char *expr) {
    int operand_stack[MAX_EXPR_LENGTH];
    int operator_stack[MAX_EXPR_LENGTH];
    int operand_top = -1;
    int operator_top = -1;

    int i = 0;
    while(expr[i] != '\0') {
        if(expr[i] >= '0' && expr[i] <= '9') {
            int operand = 0;
            while(expr[i] >= '0' && expr[i] <= '9') {
                operand = (operand * 10) + (expr[i] - '0');
                i++;
            }
            operand_stack[++operand_top] = operand;
        } else if(expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/' || expr[i] == '^') {
            while(operator_top >= 0 && precedence(operator_stack[operator_top]) >= precedence(expr[i]))
{
                evaluate(operator_stack[operator_top], operand_stack, &operand_top);
                operator_top--;
            }
            operator_stack[++operator_top] = expr[i];
            i++;
        } else {
            printf("Invalid character in expression!\n");
            exit(0);
        }
    }

    while(operator_top >= 0) {
        evaluate(operator_stack[operator_top], operand_stack, &operand_top);
        operator_top--;
    }

    printf("Result: %d\n", operand_stack[operand_top]);
}

int main() {
    char expr[MAX_EXPR_LENGTH];
```

```
    printf("Enter an expression: ");
    fgets(expr, MAX_EXPR_LENGTH, stdin);

    // Remove trailing newline from expression
    expr[strcspn(expr, "\n")] = '\0';

    operatorPrecedenceParsing(expr);

    return 0;
}
```

OUTPUT.

```
Enter an expression: 2+3*4
Result: 14
```