Yendluri Sathvika

CH.EN.U4CSE22164

COMPILER DESIGN– 19CSE401

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

Compiler Design 19CSE401

Lab Report

Experiment No.	Date	Programs
1	20-Jul-2025	Program to Identify Vowels and Consonants
2	25-Jul-2025	Program to Count Lines, Words, and Characters
3	30-Jul-2025	Program to Recognize Integers and Floating-Point Numbers
4	04-Aug-2025	Program to Recognize C Keywords
5	09-Aug-2025	Program to Recognize Operators
6	14-Aug-2025	To implement Lexical Analyzer Using Lex Tool
7	19-Aug-2025	Program to eliminate left recursion and factoring from the given grammar
8	24-Aug-2025	Program to eliminate left recursion and factoring from the given grammar
9	29-Aug-2025	To write a program in YACC for parser generation

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

10	03-Sep-2025	To implement Symbol Table
11	08-Sep-2025	To implement intermediate code generation
12	13-Sep-2025	To implementation of Code Optimization Techniques
13	18-Sep-2025	To write a program that implements the target code generation

Basic Programs

1. **Aim:** Program to Identify Vowels and Consonants

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Specify the header file <stdio.h> between %{ and %}.
- Define the character patterns for vowels [aAeEiIoOuU], alphabets [a-zA-Z], whitespaces [\t\n], and other characters ..
- Use translation rules to print whether the character is a vowel, consonant, or not an alphabet character.
- Call yylex() inside the main() function to begin lexical analysis.
- Save the program as vowelconsonant.l using the LEX language.
- Run the program using the LEX compiler to generate lex.yy.c.
- The generated lex.yy.c contains tables and routines to match input characters.
- Compile lex.yy.c using a C compiler to create an executable file.
- Run the executable to check each character in the input and classify it.

Code:

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

%%

[aAeEiIoOuU]        { printf("%s is a VOWEL\n", yytext); }
[a-zA-Z]            { printf("%s is a CONSONANT\n", yytext); }
[ \t\n]             ;  // Ignore whitespace
.                   { printf("%s is not an alphabet character\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex Q1.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lex.yy.c -o Q1 -ll
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./Q1
Enter the string.. at end press ^d
HELLOWORLD

No of vowels = 3
No of consonants = 7
```

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

Result: The program has been executed successfully.

2. **Aim:** Program to Count Lines, Words, and Characters

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Declare and initialize line, word, and character counters.
- Define regular expressions for newline, whitespace, and words.
- Use translation rules to update the respective counters.
- Call yylex() inside the main() function.
- Print the final count of lines, words, and characters.
- Save the program as counter.l.
- Run the program using the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to produce the executable.
- Run the executable to perform the counting operation on input.

Code:

```
%{
#include <stdio.h>
int lines = 0, words = 0, chars = 0;
%}

%%

\n              { lines++; chars++; }
[ \t]+          { chars += yyleng; }
[^ \t\n]+       { words++; chars += yyleng; }

%%

int main() {
    yylex();
    printf("\nLines: %d\nWords: %d\nCharacters: %d\n", lines, words, chars);
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ nano count_all.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ nano Q2.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex Q2.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lex.yy.c -o Q2 -ll
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./Q2 input.txt
Total characters: 32
Total words: 12
Total spaces/tabs: 7
Total lines: 2
```

Result: The program has been executed successfully.

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

3. **Aim:** Program to Recognize Integers and Floating-Point Numbers

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Define patterns for floating point numbers, integers, whitespaces, and other characters.
- Use translation rules to identify and print whether input is float, integer, or not a number.
- Ignore whitespaces like tab, space, and newline.
- Call yylex() inside the main() function to start lexical analysis.
- Save the program as numcheck.l.
- Run the program using the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the executable.
- Run the executable to test inputs and identify the type of number.

Code:

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

%%

[0-9]+\.[0-9]+      { printf("%s is a FLOATING POINT number\n", yytext); }
[0-9]+              { printf("%s is an INTEGER\n", yytext); }
[ \t\n]             ; // Ignore whitespace
.                   { printf("%s is not a number\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@ase-computer-lab:~/Documents$ flex q3.l
asecomputerlab@ase-computer-lab:~/Documents$ gcc lex.yy.c -ll -o scanner
asecomputerlab@ase-computer-lab:~/Documents$ ./scanner
57.90
57.90 is a FLOATING POINT number
23
23 is an INTEGER
12
12 is an INTEGER
24
24 is an INTEGER
```

Result: The program has been executed successfully.

4. **Aim:** Program to Recognize C Keywords .

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Define regular expressions for C keywords, identifiers, whitespaces, and other characters.
- Use translation rules to print whether input is a C keyword, identifier, or something else.
- Ignore spaces, tabs, and newline characters.
- Call yylex() in the main() function to begin lexical analysis.
- Save the program as keywordid.l.
- Run the program through the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the final executable.
- Run the executable to classify each token as keyword, identifier, or other.

Code:

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

%%

"int" |
"float" |
"return" |
"if" |
"else" |
"while" |
"for"              { printf("%s is a C keyword\n", yytext); }

[a-zA-Z_][a-zA-Z0-9_]* { printf("%s is an identifier\n", yytext); }

[ \t\n]            ; // Ignore spaces

.                  { printf("%s is something else\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@ase-computer-lab:~/Documents$ flex q4.l
asecomputerlab@ase-computer-lab:~/Documents$ gcc lex.yy.c -ll -o scanner
asecomputerlab@ase-computer-lab:~/Documents$ ./scanner
for
for is a C keyword
is
is is an identifier
```

Result: The program has been executed successfully.

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

5. **Aim:** Program to Recognize Operators .

Algorithm:

- Open the gedit text editor from Accessories under Applications menu.
- Include the header file <stdio.h> between %{ and %}.
- Define regular expressions for relational operators, arithmetic/assignment operators, whitespaces, and other characters.
- Use translation rules to check and print whether input is a relational operator, arithmetic/assignment operator, or not an operator.
- Ignore whitespaces like tab and newline characters.
- Call yylex() inside the main() function to begin lexical analysis.
- Save the program as operatorcheck.l.
- Run the program through the LEX compiler to generate lex.yy.c.
- Compile lex.yy.c using a C compiler to get the executable.
- Run the executable to test and classify the input operators.

Code:

```
%{
#include <stdio.h>
%}
%%
|
"==" |
"!=" |
"<=" |
">=" |
"<" |
">"                    { printf("%s is a relational operator\n", yytext); }

"+" |
"-" |
"*" |
"/" |
"="                    { printf("%s is an arithmetic/assignment operator\n", yytext); }

[ \t\n]               ; // Ignore spaces

.                     { printf("%s is not an operator\n", yytext); }

%%

int main() {
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@ase-computer-lab:~/Documents$ flex q5.l
asecomputerlab@ase-computer-lab:~/Documents$ gcc lex.yy.c -ll -o scanner
asecomputerlab@ase-computer-lab:~/Documents$ ./scanner
%
% is not an operator
>
> is a relational operator
>=
>= is a relational operator
```

Result: The program has been executed successfully.

EXPERIMENT NO – 1

Aim: To implement Lexical Analyzer Using Lex Tool

Algorithm:

- Open gedit text editor from Accessories in Applications.
- Specify the header files to be included inside the declaration part (i.e. between %{ and %}).
- Define the digits 0-9 and identifiers a-z and A-Z.
- Using translation rules, define the regular expressions for digit, keywords, identifiers, operators, header files etc. If matched with the input, store and display using yytext.
- Inside procedure main (), use yyin() to point to the current file being passed by the lexer.
  • The specification of the lexical analyzer is prepared by creating a program lab1.l in the LEX language.
- The lab1.l program is run through the LEX compiler to produce equivalent C code named lex.yy.c.
- The program lex.yy.c consists of a table constructed from the regular expressions of lab1.l, along with standard routines that use the table to recognize lexemes.
- Finally, the lex.yy.c program is run through a C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

Code:

Lab1.l:

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

int COMMENT = 0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.*                     { printf("\n%s is a preprocessor directive", yytext); }

int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto                    { printf("\n\t%s is a keyword", yytext); }

"/*"                    { COMMENT = 1; printf("\n\t%s is a COMMENT", yytext); }

{identifier}\(          { if (!COMMENT) printf("\nFUNCTION \n\t%s", yytext); }

\{                      { if (!COMMENT) printf("\n BLOCK BEGINS"); }

\}                      { if (!COMMENT) printf("BLOCK ENDS "); }

{identifier}(\[[0-9]*\])? { if (!COMMENT) printf("\n %s IDENTIFIER", yytext); }

\".*\"                  { if (!COMMENT) printf("\n\t%s is a STRING", yytext); }
```

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

```
[0-9]+                  { if (!COMMENT) printf("\n %s is a NUMBER", yytext); }

\)(\:)?                 { if (!COMMENT) { printf("\n\t"); ECHO; printf("\n"); } }

\(                      { ECHO; }

=                       { if (!COMMENT) printf("\n\t%s is an ASSIGNMENT OPERATOR", yytext); }

\<= |
\>= |
\< |
== |
\>                      { if (!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR", yytext); }

%%

int main(int argc, char **argv)
{
    FILE *file;
    file = fopen("var.c", "r");
    if (!file)
    {
        printf("Could not open the file\n");
        exit(0);
    }

    yyin = file;
    yylex();
    printf("\n");
    return 0;
}

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

Var.c:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Output:

```
asecomputerlab@lab:~$ flex lexcode.l
asecomputerlab@lab:~$ gcc lex.yy.c -o lexprogram
asecomputerlab@lab:~$ ./lexprogram

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

 c IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
         "Sum:%d" is a STRING,
 c IDENTIFIER
        )
;
BLOCK ENDS
```

Result: The program has been executed successfully.

EXPERIMENT NO – 2

Aim: Program to eliminate left recursion and factoring from the given grammar

Algorithm:

- Open any text editor and start writing a C program.
- Include the necessary header files: stdio.h and string.h.
- Declare required character arrays for grammar parts and variables for loop counters and positions.
- Prompt the user to enter a production in the form A->alpha|beta.
- Use fgets() to read the entire input line, removing the trailing newline.
- Extract the portion before the | into part1 and the portion after into part2.

- Find the longest common prefix between part1 and part2 and store it in modifiedGram.
- After the common part, append 'X' to modifiedGram to denote the new non-terminal.
- Create newGram to store the restructured productions from the remaining suffixes of part1 and part2.
- Display the final left-factored productions using printf().

Code:

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

int main() {
    char gram[50], part1[25], part2[25];
    char modifiedGram[25], newGram[25];
    int i, j = 0, k = 0, pos = 0;

    printf("Enter Production (A->): ");
    fgets(gram, sizeof(gram), stdin);  // safe input

    // Remove newline if present
    gram[strcspn(gram, "\n")] = '\0';

    // Split into part1 and part2 at '|'
    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';

    // Find common prefix
    for (i = 0; part1[i] != '\0' && part2[i] != '\0'; i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k++] = part1[i];
            pos = i + 1;
        } else {
            break;
        }
    }

    // Extract differing parts into newGram
    j = 0;
    for (i = pos; part1[i] != '\0'; i++)
        newGram[j++] = part1[i];
    newGram[j++] = '|';
    for (i = pos; part2[i] != '\0'; i++)
        newGram[j++] = part2[i];
    newGram[j] = '\0';
```

Output:

```
asecomputerlab@lab:~$ nano leftfactoring.c
asecomputerlab@lab:~$ gcc leftfactoring.c -o leftfactoring
asecomputerlab@lab:~$ ./leftfactoring
Enter Production : A->abcde|abxyz

A->abX
X->cde|xyz
```

Result: The program has been executed successfully.

EXPERIMENT NO – 2

AIM: Left recursion

Algorithm:

1. Start the processes by getting the grammar and assigning it to the appropriate variables.
2. Check if the given grammar has left recursion.
3. Identify the alpha and beta elements in the production.
4. Print the output according to the formula to remove left recursion

CODE:

```
#include <stdio.h>
#include <string.h>
#define SIZE 10

int main() {
    char non_terminal;
    char alpha[SIZE], beta[SIZE];
    int num;
    char production[10][SIZE];

    printf("Enter Number of Productions: ");
    scanf("%d", &num);
    printf("Enter the grammar (e.g., E->E-A or E->Ea|b):\n");

    for (int i = 0; i < num; i++) {
        scanf("%s", production[i]);
    }

    for (int i = 0; i < num; i++) {
        printf("\nGRAMMAR: %s", production[i]);
        non_terminal = production[i][0];

        if (production[i][3] == non_terminal) {
            // Left recursion detected
            int j = 4, k = 0;
            while (production[i][j] != '\0' && production[i][j] != '|') {
                alpha[k++] = production[i][j++];
            }
            alpha[k] = '\0';

            if (production[i][j] == '|') {
                j++; // skip '|'
                k = 0;
                while (production[i][j] != '\0') {
                    beta[k++] = production[i][j++];
                }
                beta[k] = '\0';

                printf("\nLeft recursion detected.");
                printf("\nGrammar without left recursion:\n");
                printf("%c -> %s%c'\n", non_terminal, beta, non_terminal);
                printf("%c' -> %s%c' | ε\n", non_terminal, alpha, non_terminal);
                 printf("%c' -> %s%c' | ε\n", non_terminal, alpha, non_terminal);
             } else {
                 printf("\nOnly left recursive production found. Cannot be simplified without alternate (β).");
             }
        } else {
            printf("\nNo left recursion.");
        }
    }

    return 0;
}
```

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

OUTPUT:

```
asecomputerlab@lab:~$ nano leftrecursion.c
asecomputerlab@lab:~$ gcc leftrecursion.c -o leftrecursion
asecomputerlab@lab:~$ ./leftrecursion
Enter Number of Production : 2
Enter the grammar as E->E-A :
E->E-A
E->b

GRAMMAR : : : E->E-A is left recursive.
 can't be reduced

GRAMMAR : : : E->b is not left recursive.
asecomputerlab@lab:~$ nano leftrecursion.c
asecomputerlab@lab:~$ gcc leftrecursion.c -o leftrecursion
asecomputerlab@lab:~$ ./leftrecursion
Enter Number of Production : 1
Enter the grammar as E->E-A :
E->E-A|b

GRAMMAR : : : E->E-A|b is left recursive.
Grammar without left recursion:
E->bE'
E'->-E'|E
asecomputerlab@lab:~$
```

RESULT:

Thus, the program has been successfully executed.

EXPERIMENT NO – 3

Aim: To implement LL(1) parsing using C program.

Algorithm:

- Initialize parsing table `m[][][]` and size table `size[][]`.
- Read input string from user and append `'$'` at the end.
- Initialize stack with `'$'` at the bottom and push start symbol `'e'`.
- Print header for stack and input.
- Repeat until both stack top and input symbol are not `'$'`:
- If stack top equals input symbol, pop the stack and advance input.
- Otherwise, determine row index from stack top.
- Determine column index from current input symbol.
- If no production rule exists in table, print error and exit.

- If rule is epsilon (n), pop the stack.
- If rule is a terminal like i, replace stack top with that terminal.
- Otherwise, push the right-hand side of the production rule (in reverse order) onto the stack.
- Print current contents of stack and input string.
- Continue until parsing ends.
- If successful, print "SUCCESS".

Code:

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

char input[50], stack[50];

// Parsing table
// Rows: e=0, b=1, t=2, c=3, f=4
// Columns: i=0, +=1, *=2, ( =3, )=4, $=5

char m[5][6][5] = {
    {"tb",  "",    "",    "tb", "",   ""},    // e
    {"",    "+tb", "",    "",   "n",  "n"},   // b
    {"fc",  "",    "",    "fc", "",   ""},    // t
    {"",    "n",   "*fc","",    "n",  "n"},   // c
    {"i",   "",    "",    "(e)", "",  ""}     // f
};

int size[5][6] = {
    {2, 0, 0, 2, 0, 0},  // e
    {0, 3, 0, 0, 1, 1},  // b
    {2, 0, 0, 2, 0, 0},  // t
    {0, 1, 3, 0, 1, 1},  // c
    {1, 0, 0, 3, 0, 0}   // f
};

int main() {
    int top = 1;  // stack top index
    int i = top, j = 0, k;
    int row, col;

    printf("Enter the input string: ");
    scanf("%s", input);
    strcat(input, "$");

    int len = strlen(input);

    stack[0] = '$';
    stack[1] = 'e';

    printf("\nStack\t\tInput\n");
    printf("------------------------\n");
```

```
    while (stack[i] != '$' || input[j] != '$') {
        // Print stack
        for (k = 0; k <= i; k++) printf("%c", stack[k]);
        printf("\t\t");
        // Print remaining input
        for (k = j; k < len; k++) printf("%c", input[k]);
        printf("\n");

        if (stack[i] == input[j]) {
            // Terminal match - pop and advance input
            i--;
            j++;
        }
        else {
            // Determine row from stack[i]
            switch(stack[i]) {
                case 'e': row = 0; break;
                case 'b': row = 1; break;
                case 't': row = 2; break;
                case 'c': row = 3; break;
                case 'f': row = 4; break;
                default:
                    printf("\nERROR: Invalid symbol '%c' on stack\n", stack[i]);
                    exit(0);
            }

            // Determine column from input[j]
            switch(input[j]) {
                case 'i': col = 0; break;
                case '+': col = 1; break;
                case '*': col = 2; break;
                case '(': col = 3; break;
                case ')': col = 4; break;
                case '$': col = 5; break;
                default:
                    printf("\nERROR: Invalid input symbol '%c'\n", input[j]);
                    exit(0);
            }

            if (m[row][col][0] == '\0') {
                printf("\nERROR: No rule for %c on input %c\n", stack[i], input[j]);
                exit(0);
```

```
                    exit(0);
            }

            if (m[row][col][0] == '\0') {
                printf("\nERROR: No rule for %c on input %c\n", stack[i], input[j]);
                exit(0);
            }
            else if (m[row][col][0] == 'n') {
                // epsilon production: pop non-terminal
                i--;
            }
            else {
                // Pop non-terminal
                i--;
                // Push RHS of production in reverse order
                for (k = size[row][col] - 1; k >= 0; k--) {
                    stack[++i] = m[row][col][k];
                }
            }
        }
    }

    printf("\nSUCCESS: String parsed successfully!\n");

    return 0;
}
```

Output:

```
asecomputerlab@ase-computer-lab:~/Documents$ gcc ll1parser.c -o ll1parser
asecomputerlab@ase-computer-lab:~/Documents$ ./ll1parser
Enter the input string: i+i*i

Stack           Input
-----------------------
$e              i+i*i$
$bt             i+i*i$
$bcf            i+i*i$
$bci            i+i*i$
$bc             +i*i$
$b              +i*i$
$bt+            +i*i$
$bt             i*i$
$bcf            i*i$
$bci            i*i$
$bc             *i$
$bcf*           *i$
$bcf            i$
$bci            i$
$bc             $
$b              $

SUCCESS: String parsed successfully!
```

Result: The program has been executed successfully.

EXPERIMENT NO – 4

Aim: To write a program in YACC for parser generation.

Algorithm:

- Start program and define grammar tokens (NUMBER, operators, parentheses) and their precedence.
- Accept input lines containing arithmetic expressions.
- Parse the expression according to grammar rules (+, -, *, /, parentheses, unary minus, numbers).
- Perform arithmetic operations as semantic actions during parsing.
- Use yylex() to read input, skip spaces, and return tokens (numbers or operators).
- When a number is found, read it fully and assign to yylval.
- Continue parsing until the entire expression is reduced.
- Print the evaluated result of the expression and repeat for next input.

Code:

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

int yylex();
void yyerror(const char *s);
%}

%union {
    double val;
}

%token <val> NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%type <val> expr

%%
lines:
    lines expr '\n'   { printf("= %g\n", $2); }
  | lines '\n'
  | /* empty */
  ;

expr:
    expr '+' expr     { $$ = $1 + $3; }
  | expr '-' expr     { $$ = $1 - $3; }
  | expr '*' expr     { $$ = $1 * $3; }
  | expr '/' expr     {
                        if ($3 == 0) {
                            yyerror("Division by zero");
                            YYABORT;
                        }
                        $$ = $1 / $3;
                      }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | '(' expr ')'      { $$ = $2; }
  | NUMBER            { $$ = $1; }
```

```
%{
#include "y.tab.h"
%}

%%
[ \t]+                ; // Skip spaces and tabs
[0-9]+(\.[0-9]+)?     {
                        yylval.val = atof(yytext);
                        return NUMBER;
                      }
\n                    { return '\n'; }
.                     { return yytext[0]; }
%%

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

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Documents$ bison -d calc.y
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Documents$ gcc lex.yy.c calc.tab.c -o calc -lm
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~/Documents$ ./calc
Enter expressions (Ctrl+D to quit):
(8 + 2) * 3
= 30
```

Result: The program has been executed successfully.

EXPERIMENT NO – 5

Aim: To implement Symbol Table.

Algorithm:

- Start the program and read an expression ending with $.
- Store the input characters into an array.
- Display the given expression.
- Traverse each character of the expression.
- If the character is an alphabet, classify it as an identifier and store with its address.
- If the character is an operator (+, -, *, =), classify it as an operator and store with its address.
- Display the complete symbol table and end the program.

Code:

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

int main() {
    int x = 0, n, i = 0, j = 0;
    void *mypointer, *T4Tutorials_address[15];
    char ch, c;
    char T4Tutorials_Array2[15], T4Tutorials_Array3[15];

    printf("Input the expression ending with $ sign: ");
    while ((c = getchar()) != '$' && i < 15) {
        T4Tutorials_Array2[i] = c;
        i++;
    }
    n = i - 1;

    printf("Given Expression: ");
    i = 0;
    while (i <= n) {
        printf("%c", T4Tutorials_Array2[i]);
        i++;  // <-- semicolon added here
    }

    printf("\n\nSymbol Table display\n");
    printf("Symbol \t addr \t type\n");

    j = 0;
    while (j <= n) {
        c = T4Tutorials_Array2[j];
        if (isalpha((unsigned char)c)) {
            mypointer = malloc(1);
            T4Tutorials_address[x] = mypointer;
            T4Tutorials_Array3[x] = c;
            printf("%c \t %p \t identifier\n", c, mypointer);  // use %p for pointers
```

```
            x++;
            j++;  // <-- semicolon added here
        } else {
            ch = c;
            if (ch == '+' || ch == '-' || ch == '*' || ch == '=') {
                mypointer = malloc(1);
                T4Tutorials_address[x] = mypointer;
                T4Tutorials_Array3[x] = ch;
                printf("%c \t %p \t operator\n", ch, mypointer);  // use %p here too
                x++;
                j++;  // <-- semicolon added here
            } else {
                j++;  // <-- semicolon added here
            }
        }
    }

    // Free allocated memory
    for (i = 0; i < x; i++) {
        free(T4Tutorials_address[i]);
    }

    return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lab5.c -o lab5
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./lab5
Input the expression ending with $ sign: w=a+b*c$
Given Expression: w=a+b*c

Symbol Table display
Symbol   addr     type
w        0x55bb02cf5a80          identifier
=        0x55bb02cf5aa0          operator
a        0x55bb02cf5ac0          identifier
+        0x55bb02cf5ae0          operator
b        0x55bb02cf5b00          identifier
*        0x55bb02cf5b20          operator
c        0x55bb02cf5b40          identifier
```

Result: The program has been executed successfully.

EXPERIMENT NO – 6

Aim: To implement intermediate code generation.

Algorithm:

- Start the program and read an arithmetic expression as input.
- Scan the expression and record the positions of operators (:, /, *, +, -).
- For each operator, find its left operand and right operand.
- Generate a temporary variable for the result and replace the operator with it.

- Print the intermediate code in the form of three-address statements (T := operand1 op operand2).
- Repeat the process until the full expression is reduced.
- Print the final assignment statement and end the program.

Code:

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

int i = 1, j = 0, no = 0, tmpch = 90;
char str[100], left[15], right[15];

void findopr();
void explore();
void fleft(int);
void fright(int);

struct exp {
    int pos;
    char op;
} k[15];

int main() {
    printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
    printf("Enter the Expression: ");
    scanf("%s", str);
    printf("\nThe Intermediate Code:\n");
    findopr();
    explore();
    return 0;
}

// Function to find operator positions in the expression
void findopr() {
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == ':') {
            k[j].pos = i;
            k[j++].op = ':';
        }
    }
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '/') {
            k[j].pos = i;
            k[j++].op = '/';
        }
    }
```

```
        }
        x--;
    }
}

// Function to find right operand
void fright(int x) {
    int w = 0, flag = 0;
    x++;

    while (x != -1 && str[x] != '+' && str[x] != '-' && str[x] != '*' && str[x] != '/' &&
           str[x] != '=' && str[x] != ':' && str[x] != '\0') {
        if (str[x] != '$' && flag == 0) {
            right[w++] = str[x];
            right[w] = '\0';
            str[x] = '$'; // Mark as used
            flag = 1;
        }
        x++;
    }
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lab6.c -o lab6
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./lab6
                INTERMEDIATE CODE GENERATION

Enter the Expression: w=a+b*c
The intermediate code:
        Z := w = a
        Y := w + b
        X := w * c
        w := X
```

Result: The program has been executed successfully.

EXPERIMENT NO – 7

Aim: To implementation of Code Optimization Techniques

Algorithm:

- Start the program and read the number of expressions (n).
- For each expression, input the left-hand side variable and the right-hand side expression.

- Display the original intermediate code.
- Perform dead code elimination by keeping only those statements whose results are used later.
- Perform common subexpression elimination by checking if two expressions compute the same value and replacing duplicates.
- Update references so that redundant variables are replaced with the optimized variable.
- Print the final optimized code and end the program.

Code:

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

struct op {
    char l;
    char r[20];
} op[10], pr[10];

int main() {
    int a, i, k, j, n, z = 0, m, q;
    char *p, *l;
    char temp, t;
    char *tem;

    printf("Enter the Number of Values: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Left: ");
        scanf(" %c", &op[i].l);
        printf("Right: ");
        scanf(" %s", op[i].r);
    }

    // Print intermediate code
    printf("\nIntermediate Code\n");
    for (i = 0; i < n; i++) {
        printf("%c = %s\n", op[i].l, op[i].r);
    }

    // Dead code elimination: find used expressions
    for (i = 0; i < n - 1; i++) {
        temp = op[i].l;
        for (j = 0; j < n; j++) {
            p = strchr(op[j].r, temp);
            if (p) {
                pr[z].l = op[i].l;
                strcpy(pr[z].r, op[i].r);
                z++;
                break; // only once per use
            }
        }
    }
```

```c
pr[z].l = op[n - 1].l;
strcpy(pr[z].r, op[n - 1].r);
z++;

printf("\nAfter Dead Code Elimination\n");
for (k = 0; k < z; k++) {
    printf("%c = %s\n", pr[k].l, pr[k].r);
}

// Common subexpression elimination (substitute reused RHS)
for (m = 0; m < z; m++) {
    tem = pr[m].r;
    for (j = m + 1; j < z; j++) {
        p = strstr(tem, pr[j].r);
        if (p) {
            t = pr[j].l;
            pr[j].l = pr[m].l;
            for (i = 0; i < z; i++) {
                l = strchr(pr[i].r, t);
                if (l) {
                    a = l - pr[i].r;
                    pr[i].r[a] = pr[m].l;
                }
            }
        }
    }
}

// Print code after common subexpression elimination
printf("\nAfter Common Subexpression Elimination\n");
for (i = 0; i < z; i++) {
    printf("%c = %s\n", pr[i].l, pr[i].r);
}

// Remove duplicates (fully redundant expressions)
for (i = 0; i < z; i++) {
    for (j = i + 1; j < z; j++) {
        q = strcmp(pr[i].r, pr[j].r);
        if ((pr[i].l == pr[j].l) && q == 0) {
            pr[j].l = '\0'; // mark for deletion
        }
    }
}


// Final optimized code
printf("\nOptimized Code\n");
for (i = 0; i < z; i++) {
    if (pr[i].l != '\0') {
        printf("%c = %s\n", pr[i].l, pr[i].r);
    }
}

return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lab7.c -o lab7
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./lab7
Enter the Number of Values: 3
left: a
right: b+c
left: d
right: a*e
left: f
right: d+g

Intermediate Code
a = b+c
d = a*e
f = d+g

After Dead Code Elimination
a       = b+c
d       = a*e
f       = d+g

After Eliminating Common Expressions
a       = b+c
d       = a*e
f       = d+g

Optimized Code
a = b+c
d = a*e
f = d+g
```

Result: The program has been executed successfully.

EXPERIMENT NO – 8

Aim: To write a program that implements the target code generation **Algorithm:**

- Read the input string from the user.
- Process each input string and use a switch–case structure to identify the operator.
- Load the input variables into temporary variables (operands) and display them using the instruction LOAD.
- Based on the arithmetic operator, display the corresponding operation (ADD, SUB, MUL, DIV) using switch–case.
- Generate the three-address code representation for each operation.
- If the operator is an assignment (=), store the result in the target variable and display it using STORE.
- Repeat this process for each line of the input string.
- Display the final output, which is the transformed assembly-like machine code.

Code:

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

int label[20];
int no = 0;

int check_label(int k);

int main() {
    FILE *fp1, *fp2;
    char fname[20], op[10], ch;
    char operand1[8], operand2[8], result[8];
    int i = 0, j = 0;

    printf("\nEnter filename of the intermediate code: ");
    scanf("%s", fname);

    fp1 = fopen(fname, "r");
    fp2 = fopen("target.txt", "w");

    if (fp1 == NULL || fp2 == NULL) {
        printf("\nError opening the file\n");
        exit(0);
    }

    while (fscanf(fp1, "%s", op) != EOF) {
        i++;
        if (check_label(i))
            fprintf(fp2, "\nlabel#%d\n", i);

        if (strcmp(op, "print") == 0) {
            fscanf(fp1, "%s", result);
            fprintf(fp2, "\tOUT %s\n", result);
        }
        else if (strcmp(op, "goto") == 0) {
            fscanf(fp1, "%s %s", operand1, operand2);
            fprintf(fp2, "\tJMP %s,label#%s\n", operand1, operand2);
            label[no++] = atoi(operand2);
        }
        else if (strcmp(op, "[]=") == 0) {
            fscanf(fp1, "%s %s %s", operand1, operand2, result);
            fprintf(fp2, "\tSTORE %s[%s],%s\n", operand1, operand2, result);
        }

            fprintf(fp2, "\tSTORE R1,%s\n", result);
        }
        else {
            // Switch on first character of the op string
            switch (op[0]) {
                case '*':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s,R1\n", operand2);
                    fprintf(fp2, "\tMUL R1,R0\n");
                    fprintf(fp2, "\tSTORE R0,%s\n", result);
                    break;

                case '+':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s,R1\n", operand2);
                    fprintf(fp2, "\tADD R1,R0\n");
                    fprintf(fp2, "\tSTORE R0,%s\n", result);
                    break;

                case '-':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s,R1\n", operand2);
                    fprintf(fp2, "\tSUB R1,R0\n");
                    fprintf(fp2, "\tSTORE R0,%s\n", result);
                    break;

                case '/':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s,R1\n", operand2);
                    fprintf(fp2, "\tDIV R1,R0\n");
                    fprintf(fp2, "\tSTORE R0,%s\n", result);
                    break;

                case '%':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s,R1\n", operand2);
                    fprintf(fp2, "\tDIV R1,R0\n");
                    fprintf(fp2, "\tSTORE R0,%s\n", result);
                    break;
```

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171

```c
                fscanf(fp1, "%s %s", operand1, result);
                fprintf(fp2, "\tSTORE %s,%s\n", operand1, result);
                break;

            case '>':
                fscanf(fp1, "%s %s %s", operand1, operand2, result);
                fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                fprintf(fp2, "\tJGT %s,label#%s\n", operand2, result);
                label[no++] = atoi(result);
                break;

            case '<':
                fscanf(fp1, "%s %s %s", operand1, operand2, result);
                fprintf(fp2, "\tLOAD %s,R0\n", operand1);
                fprintf(fp2, "\tJLT %s,label#%s\n", operand2, result);
                label[no++] = atoi(result);
                break;

            default:
                // Handle unknown operation or skip
                break;
            }
        }
    }

    fclose(fp1);
    fclose(fp2);

    // Display generated target code
    fp2 = fopen("target.txt", "r");
    if (fp2 == NULL) {
        printf("Error opening the target file\n");
        exit(0);
    }

    while ((ch = fgetc(fp2)) != EOF) {
        putchar(ch);
    }

    fclose(fp2);

    return 0;
}

int check_label(int k) {
    for (int i = 0; i < no; i++) {
        if (k == label[i]) return 1;
    }
    return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lab8.c -o lab8
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./lab8

Enter filename of the intermediate code: input.txt
        LOAD 34,R0
        LOAD +45,R1
        SUB R1,R0
        STORE R0,-6.78
        LOAD +7.56,R0
        LOAD 99,R1
        SUB R1,R0
        STORE R0,-6.78
```

Result: The program has been executed successfully.

NAME: B.MOKSHITHA
ROLLNO: CH.EN.U4CSE22171