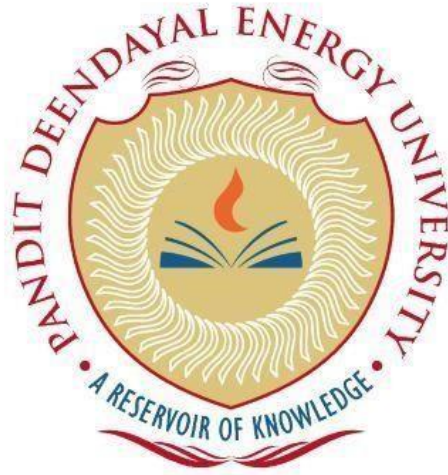


**PANDIT DEENDAYAL ENERGY UNIVERSITY**  
**SCHOOL OF TECHNOLOGY**



**Course: System Software and Compiler Design Lab**

**Course Code: 20CP302P**

**LAB MANUAL**

**B.Tech. (Computer Science and Engineering)**

**Semester 5**

**Submitted By:**

Harsh Shah

21BCP359

G11 batch

<b>Sr. No.</b>	<b>Index</b>	<b>Page No.</b>
<b>1</b>	Write C/C++ program to identify keywords, identifiers and others from the given input file.	<b>2</b>
<b>2</b>	a. Write a LEX program to count the number of tokens and display each token with its length in the given statements. Write a LEX program to identify keywords, identifiers, numbers and other b. characters and generate tokens for each.	<b>5</b>
<b>3</b>	a. Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c. Write a LEX program to count the number of characters, words and lines in the given input. Write a LEX program that read the numbers and add 3 to the numbers if the b. number is divisible by 7.	<b>9</b>
<b>4</b>	WAP to implement Recursive Decent Parser (RDP) parser for given grammar.	<b>13</b>
<b>5</b>	Write a program to calculate first and follow of a given LL (1) grammar.	<b>18</b>
<b>6</b>	WAP to construct operator precedence parsing table for the given grammar and check the validity of the string.	<b>23</b>
<b>7</b>	a. Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, *, / and ↑). b. Write a YACC program for desktop calculator with ambiguous grammar and additional information. c. Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules: $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow P \uparrow F \mid P$ $P \rightarrow (E) \mid id$ And parse the sentence: id + id * id.	<b>26</b>
<b>9</b>	Implement menu driven program to execute any 2 code optimization techniques on given code.	<b>32</b>
<b>10</b>	Select one block or expression from C language and generate symbol table and target code for the same.	<b>34</b>

## Experiment – 1

### Aim:

Write C/C++ program to identify keywords, identifiers and others from the given input file.

### Code:

#### Lab0.c

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

```
int main(){  
    int a = 10, b = 20;  
    int sum = a + b;  
    printf("Sum = %d", sum);  
    return 0;  
}
```

#### Lab1.c

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

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

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

```
#define MAX_IDENTIFIERS 100
```

```
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"  
};
```

```
int is_keyword(const char* word) {  
    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
```

```
        if (strcmp(word, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int main() {
    char identifier[MAX_IDENTIFIERS][50];
    int numIdentifiers = 0;

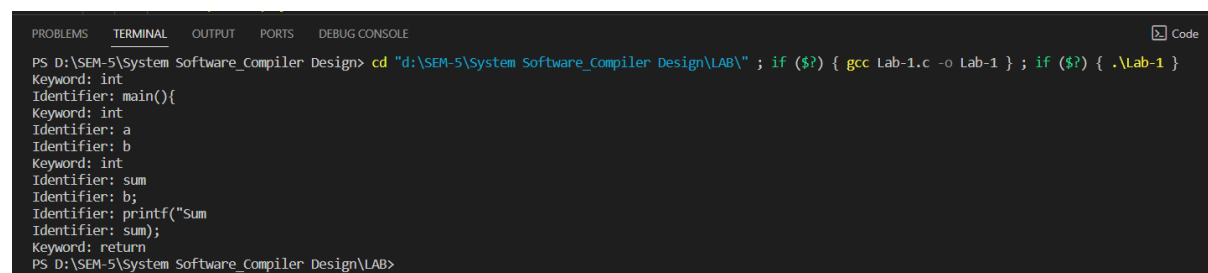
    FILE* file = fopen("Lab-0.c", "r");

    char word[50];
    while (fscanf(file, "%s", word) != EOF) {
        if (is_keyword(word) == 1) {
            printf("Keyword: %s\n", word);
        } else if (isalpha(word[0])) {
            int found = 0;
            for (int i = 0; i < numIdentifiers; i++) {
                if (strcmp(word, identifier[i]) == 0) {
                    found = 1;
                    break;
                }
            }

            if (!found) {
                strcpy(identifier[numIdentifiers], word);
                printf("Identifier: %s\n", identifier[numIdentifiers]);
                numIdentifiers++;
            }
        }
    }
}
```

```
        if (numIdentifiers == MAX_IDENTIFIERS) {  
            printf("Maximum number of identifiers reached.\n");  
            break;  
        }  
    }  
}  
}  
}  
}  
  
fclose(file);  
  
return 0;  
}
```

## Output:



```
PROBLEMS  TERMINAL  OUTPUT  PORTS  DEBUG CONSOLE  Code  
PS D:\SEM-5\System Software_Compiler Design> cd "d:\SEM-5\System Software_Compiler Design\LAB\" ; if ($?) { gcc Lab-1.c -o Lab-1 } ; if ($?) { .\Lab-1 }  
Keyword: int  
Identifier: main(){  
Keyword: int  
Identifier: a  
Identifier: b  
Keyword: int  
Identifier: sum  
Identifier: b;  
Identifier: printf("Sum  
Identifier: sum);  
Keyword: return  
PS D:\SEM-5\System Software_Compiler Design\LAB>
```

## Experiment – 2

### Aim:

- Write a LEX program to count the number of tokens and display each token with its length in the given statements.
- Write a LEX program to identify keywords, identifiers, numbers and other characters and generate tokens for each.

### Code:

#### Lab3.c

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

```
int main() {
```

```
    int a, b;
```

```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &a, &b);
```

```
    int sum = a + b;
```

```
    printf("Sum: %d\n", sum);
```

```
    return 0;
```

```
}
```

#### a)

```
%option noyywrap
```

```
%{
```

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

```
%}
```

```
%%
```

```
[a-zA-Z]+ { printf("Token: %s, Length: %d\n", yytext, yyleng); }
```

```
[0-9]+ { printf("Token: %s, Length: %d\n", yytext, yyleng); }
```

```
.\n { printf("Ignoring: %s\n", yytext); }
```

```
%%
```

```
int main() {
    yyin = fopen("Lab3.c", "r");
    yylex();
    return 0;
}
```

**b)**

%option noyywrap

%{

#include <stdio.h>

%}

%%

int|float|if|else|for|while|return { printf("Keyword: %s\n", yytext); }

[a-zA-Z][a-zA-Z0-9]\* { printf("Identifier: %s\n", yytext); }

[0-9]+ { printf("Number: %s\n", yytext); }

.\|n { printf("Other: %s\n", yytext); }

%%

```
int main() {
    yyin = fopen("Lab3.c", "r");
    yylex();
    return 0;
}
```

**Output:**

**a)**

```
PS D:\SEM-5\System Software_Compiler Design\LAB> .\a.exe
Ignoring: #
Token: include, Length: 7
Ignoring:
Ignoring: <
Token: stdio, Length: 5
Ignoring: .
Token: h, Length: 1
Ignoring: >
Ignoring:
Ignoring:
Token: int, Length: 3
Ignoring:
Token: main, Length: 4
Ignoring: (
Ignoring: )
Ignoring:
Ignoring: {
Ignoring:
Ignoring:
Ignoring:
Token: int, Length: 3
Ignoring:
Token: a, Length: 1
Ignoring: ,
Ignoring:
Token: b, Length: 1
Ignoring: ;
Ignoring:
Ignoring:
Ignoring:
Ignoring:
Token: printf, Length: 6
Ignoring: (
Ignoring: "
Token: Enter, Length: 5
Ignoring:
Token: two, Length: 3
```

```
Ignoring:
Token: b, Length: 1
Ignoring: ;
Ignoring:
Ignoring:
Ignoring:
Ignoring:
Token: printf, Length: 6
Ignoring: (
Ignoring: "
Token: Sum, Length: 3
Ignoring: :
Ignoring: %
Token: d, Length: 1
Ignoring: \
Token: n, Length: 1
Ignoring: "
Ignoring: ,
Ignoring:
Token: sum, Length: 3
Ignoring: )
Ignoring: ;
Ignoring:
Ignoring:
Ignoring:
Ignoring:
Token: return, Length: 6
Ignoring:
Token: 0, Length: 1
Ignoring: ;
Ignoring:
Ignoring: }
Ignoring:
```

```
PS D:\SEM-5\System Software_Compiler Design\LAB> █
```



b)

```

PS D:\SEM-5\System Software_Compiler Design\LAB> .\a.exe
Other: #
Identifier: include
Other:
Other: <
Identifier: stdio
Other: .
Identifier: h
Other: >
Other:

Other:

Keyword: int
Other:
Identifier: main
Other: (
Other: )
Other: {
Other:

Other:
Other:
Other:
Other:
Keyword: int
Other:
Identifier: a
Other: ,
Other:
Identifier: b
Other: ;
Other:

Other:
Other:
Other:
Other:
Identifier: printf

```

```

Identifier: Sum
Other: :
Other:
Other: %
Identifier: d
Other: \
Identifier: n
Other: "
Other: ,
Other:
Identifier: sum
Other: )
Other: ;
Other:

Other:
Other:
Other:
Other:
Keyword: return
Other:
Number: 0
Other: ;
Other:

Other: }
Other:

```

```

Other:
Other:
Other:
Other:
Identifier: scanf
Other: (
Other: "
Other: %
Identifier: d
Other:
Other: %
Identifier: d
Other: "
Other: ,
Other:
Other: &
Identifier: a
Other: ,
Other:
Other: &
Identifier: b
Other: )
Other: ;
Other:

Other:
Other:
Other:
Other:
Keyword: int
Other:
Identifier: sum
Other:
Other: =
Other:
Identifier: a
Other:
Other: +
Other:

```

## Experiment – 3

### Aim:

- Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c.
- Write a LEX program to count the number of characters, words and lines in the given input.
- Write a LEX program that read the numbers and add 3 to the numbers if the number is divisible by 7.

### Code:

#### a)

```
%option noyywrap
```

```
%{
```

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

```
%}
```

```
%option noyywrap
```

```
%%
```

```
"/"(.|\n)* { FILE *outFile = fopen("comments.txt", "a"); fprintf(outFile, "%s", yytext);  
fclose(outFile); }
```

```
.\n { printf("%s", yytext); }
```

```
%%
```

```
int main() {
```

```
    yyin = fopen("Lab3.c", "r");
```

```
    yylex();
```

```
    return 0;
```

```
}
```

#### b)

```
%option noyywrap
```

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

int charCount = 0;
int wordCount = 0;
int lineCount = 0;
}%

%%

.      { charCount++; }
\n     { charCount++; lineCount++; }
[a-zA-Z]+ { wordCount++; }

%%

int main() {
    yyin = fopen("Lab3.c", "r");
    yylex();
    printf("Character count: %d\n", charCount);
    printf("Word count: %d\n", wordCount);
    printf("Line count: %d\n", lineCount);
    return 0;
}
```

**c)**

```
%option noyywrap

%{
#include <stdio.h>
}%

%%

[0-9]+ {
    int num = atoi(yytext);
```

```

    if (num % 7 == 0) {
        num += 3;
    }
    printf("%d ", num);
}
.\n    { printf("%s", yytext); }
%%

```

```

int main() {
    yyin = fopen("Lab3.c", "r");
    yylex();
    return 0;
}

```

## Output:

a)

```

PS D:\SEM-5\System Software_Compiler Design> cd "D:\SEM-5\System Software_Compiler Design\LAB"
PS D:\SEM-5\System Software_Compiler Design\LAB> flex lab3a.l
PS D:\SEM-5\System Software_Compiler Design\LAB> gcc .\lex.yy.c
PS D:\SEM-5\System Software_Compiler Design\LAB> .\a.exe
#include <stdio.h>

int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    int sum = a + b;
    printf("Sum: %d\n", sum);
    return 0;
}
PS D:\SEM-5\System Software_Compiler Design\LAB>

```

b)

```

PS D:\SEM-5\System Software_Compiler Design\LAB> flex lab3b.l
PS D:\SEM-5\System Software_Compiler Design\LAB> gcc .\lex.yy.c
PS D:\SEM-5\System Software_Compiler Design\LAB> .\a.exe
Character count: 105
Word count: 16
Line count: 10
PS D:\SEM-5\System Software_Compiler Design\LAB>

```

c)

```
PS D:\SEM-5\System Software_Compiler Design\LAB> .\a.c
#include <stdio.h>

int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    int sum = a + b;
    printf("Sum: %d\n", sum);
    return 3 ;
}
PS D:\SEM-5\System Software_Compiler Design\LAB> █
```

## Experiment – 4

### Aim:

WAP to implement Recursive Decent Parser (RDP) parser for given grammar.

### Code:

```
#include <stdio.h>

#include <string.h>

#define SUCCESS 1
#define FAILED 0

// Function prototypes
int E(), Edash(), T(), Tdash(), F();

const char *cursor;
char string[64];

int main() {
    puts("Enter the string");
    scanf("%s", string); // Read input from the user
    cursor = string;
    puts("");
    puts("Input          Action");
    puts("-----");

    // Call the starting non-terminal E
    if (E() && *cursor == '\0') { // If parsing is successful and the cursor has reached the
end
        puts("-----");
        puts("String is successfully parsed");
        return 0;
    }
```

```
    }  
    else {  
        puts("-----");  
        puts("Error in parsing String");  
        return 1;  
    }  
}
```

// Grammar rule: E -> T E'

```
int E() {  
    printf("%-16s E -> T E'\n", cursor);  
    if (T()) { // Call non-terminal T  
        if (Edash()) // Call non-terminal E'  
            return SUCCESS;  
        else  
            return FAILED;  
    }  
    else  
        return FAILED;  
}
```

// Grammar rule: E' -> + T E' | \$

```
int Edash() {  
    if (*cursor == '+') {  
        printf("%-16s E' -> + T E'\n", cursor);  
        cursor++;  
        if (T()) { // Call non-terminal T  
            if (Edash()) // Call non-terminal E'  
                return SUCCESS;  
            else
```

```
        return FAILED;
    }
    else
        return FAILED;
}
else {
    printf("%-16s E' -> $\n", cursor);
    return SUCCESS;
}
}
```

// Grammar rule: T -> F T'

```
int T() {
    printf("%-16s T -> F T'\n", cursor);
    if (F()) { // Call non-terminal F
        if (Tdash()) // Call non-terminal T'
            return SUCCESS;
        else
            return FAILED;
    }
    else
        return FAILED;
}
```

// Grammar rule: T' -> \* F T' | \$

```
int Tdash() {
    if (cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) { // Call non-terminal F
```



```
        if (Tdash()) // Call non-terminal T'
            return SUCCESS;
        else
            return FAILED;
    }
    else
        return FAILED;
}
else {
    printf("%-16s T' -> $\n", cursor);
    return SUCCESS;
}
}
```

// Grammar rule:  $F \rightarrow ( E ) \mid i$

```
int F() {
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) { // Call non-terminal E
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            }
            else
                return FAILED;
        }
        else
            return FAILED;
    }
}
```

```
    else if (*cursor == 'i') {  
        printf("%-16s F -> i\n", cursor);  
        cursor++;  
        return SUCCESS;  
    }  
    else  
        return FAILED;  
}
```

### Output:

```
Enter the string  
i+i*i  
  
Input      Action  
-----  
i+i*i      E -> T E'  
i+i*i      T -> F T'  
i+i*i      F -> i  
+i*i       T' -> $  
+i*i       E' -> + T E'  
i*i        T -> F T'  
i*i        F -> i  
*i         T' -> $  
*i         E' -> $  
-----
```

## Experiment – 5

### Aim:

Write a program to calculate first and follow of a given LL (1) grammar.

### Code:

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define MAX_SYMBOLS 10

char nonTerminals[] = "SAB";

char terminals[] = "ab";

char productions[][10] = {"S->aAb", "S->B", "A->a", "A->null", "B->b"};

bool isTerminal(char symbol) {
    for (int i = 0; terminals[i]; i++) {
        if (terminals[i] == symbol) {
            return true;
        }
    }
    return false;
}

void addToSet(char set[], char symbol) {
    if (strchr(set, symbol) == NULL) {
        strncat(set, &symbol, 1);
    }
}
```

```
void calculateFirst(char nonTerminal, char first[]) {  
    for (int i = 0; productions[i][0]; i++) {  
        if (productions[i][0] == nonTerminal) {  
            int j = 3;  
            while (productions[i][j] != '\0') {  
                if (isTerminal(productions[i][j])) {  
                    addToSet(first, productions[i][j]);  
                    break;  
                }  
                else if (productions[i][j] != "null") {  
                    calculateFirst(productions[i][j], first);  
                    if (!strchr(first, "null")) {  
                        break;  
                    }  
                }  
                j++;  
            }  
        }  
    }  
}
```

```
void calculateFollow(char nonTerminal, char follow[]) {  
    if (nonTerminal == 'S') {  
        // Start symbol  
        addToSet(follow, '$');  
    }  
  
    for (int i = 0; productions[i][0]; i++) {  
        for (int j = 3; productions[i][j]; j++) {  
            if (productions[i][j] == nonTerminal) {
```

```
int k = j + 1;
while (productions[i][k] != '\0') {
    if (isTerminal(productions[i][k])) {
        addToSet(follow, productions[i][k]);
        break;
    } else {
        char first[MAX_SYMBOLS] = {0};
        calculateFirst(productions[i][k], first);
        bool epsilonFound = false;
        for (int m = 0; first[m]; m++) {
            if (first[m] == "null") {
                epsilonFound = true;
            } else {
                addToSet(follow, first[m]);
            }
        }
        if (!epsilonFound) {
            break;
        }
    }
    k++;
}
if (productions[i][k] == '\0') {
    if (productions[i][0] != nonTerminal) {
        calculateFollow(productions[i][0], follow);
    }
}
}
```

```
}

int main() {
    char first[MAX_SYMBOLS] = {0};
    char follow[MAX_SYMBOLS] = {0};

    for (int i = 0; nonTerminals[i]; i++) {
        calculateFirst(nonTerminals[i], first);
    }

    for (int i = 0; nonTerminals[i]; i++) {
        calculateFollow(nonTerminals[i], follow);
    }

    printf("\nFIRST sets:\n");
    for (int i = 0; nonTerminals[i]; i++) {
        printf("FIRST(%c) = {%s}\n", nonTerminals[i], first);
        first[0] = '\0'; // Clear FIRST set for the next non-terminal
    }

    printf("\nFOLLOW sets:\n");
    for (int i = 0; nonTerminals[i]; i++) {
        printf("FOLLOW(%c) = {%s}\n", nonTerminals[i], follow);
        follow[0] = '\0'; // Clear FOLLOW set for the next non-terminal
    }

    return 0;
}
```

**Output:**

```
FIRST sets:  
FIRST(S) = {ab}  
FIRST(A) = {}  
FIRST(B) = {}  
  
FOLLOW sets:  
FOLLOW(S) = {$b}  
FOLLOW(A) = {}  
FOLLOW(B) = {}  
PS D:\SEM-5\System Software_Compiler Design\LAB> █
```

## Experiment – 6

### Aim:

WAP to construct operator precedence parsing table for the given grammar and check the WAP to construct operator precedence parsing table for the given grammar and check the validity of string.

### Code:

```
from collections import defaultdict

# Example grammar
grammar = {
    'E': ['E + T', 'T'],
    'T': ['T * F', 'F'],
    'F': ['( E )', 'id']
}

# Extract the set of terminals and non-terminals
terminals = set()
non_terminals = set()
for lhs, rhs in grammar.items():
    non_terminals.add(lhs)
    for symbols in rhs:
        for symbol in symbols.split():
            if symbol not in non_terminals:
                terminals.add(symbol)
terminals.add('$')

# Initialize the table with empty cells
table = defaultdict(lambda: defaultdict(str))
for t1 in terminals:
    for t2 in terminals:
```



```
table[t1][t2] = ""

# Fill in the table with precedence relations
for lhs, rhs in grammar.items():
    for i in range(len(rhs)):
        symbols = rhs[i].split()
        for j in range(len(symbols) - 1):
            left = symbols[j]
            right = symbols[j + 1]
            if right in non_terminals:
                # Add all symbols that can follow the right symbol
                for follow in terminals:
                    table[right][follow] += '<'
            elif left in non_terminals:
                # Add all symbols that can precede the left symbol
                for precede in terminals:
                    table[precede][left] += '>'
            elif left == '(' and right == ')':
                # Parentheses have highest precedence
                pass
            else:
                # Compare the precedence of the two adjacent symbols
                table[left][right] += '=' if left == right == '+' or left == right == '*' else '<' if left == '+' or right == '*' else '>'

# Print the table
print(' ' * 4, end="")
for t in terminals:
    print(f'{t:^4}', end="")
print()
for t1 in terminals:
```

```
print(f'{t1:^4}', end="")
```

```
for t2 in terminals:
```

```
    print(f'{table[t1][t2]:^4}', end="")
```

```
print()
```

## Output:

```
precedence_table (1) x
C:\Python39\python.exe "C:/Users/Adit/Downloads/precedence_table (1).py"
+   F   *   id  )   $   (   T
+           >
F  <  <  <  <  <  <  <  ><
*           >
id          >
)           >
$           >
(           >
T  <  <  <  <  <  <  <>

Process finished with exit code 0
```

## Experiment – 7

### Aim:

- a) Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, \*, / and ↑).
- b) Write a YACC program for desktop calculator with ambiguous grammar and additional information.
- c) Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules:
  - a.  $E \rightarrow E + T \mid T$
  - b.  $T \rightarrow T * F \mid F$
  - c.  $F \rightarrow P \uparrow F \mid P$
  - d.  $P \rightarrow (E) \mid id$

And parse the sentence:  $id + id * id$ .

### Code:

a)

**prog.y**

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

%}

%token NAME NUMBER

%%

statement: NAME '=' expression
| expression {printf("= %d\n", $1);}
;

expression: expression '+' NUMBER { $$ = $1 + $3; }
| expression '-' NUMBER { $$ = $1 - $3; }
|      expression '*' expression      { $$ = $1 * $3; }
|      expression '/' expression      { $$ = $1 / $3; }
| NUMBER { $$ = $1; }
;

%%

void main(){ printf("\n Enter Expression: "); yyparse();
}
```

```
void yyerror(){ printf("\n Entered arithmetic Expression is Invalid\n\n");
}
```

### **prog.l**

```
%{
#include "prog1.tab.h"
#include<stdio.h> extern int
yylval;
%}

%%

[0-9]+ {yylval = atoi(yytext); return NUMBER;}
[ \t]; /*Ignore white space*/
\n return 0; /*Logical EOF*/
. return yytext[0]; /*to not return any symbol to the parser*/
%%

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

### **b)**

### **prog.y**

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

%token NAME NUMBER

%%

statement: NAME '=' expression
| expression { printf("Result: %d\n", $1); }
;

expression: expression '+' NUMBER { $$ = $1 + $3; }
| expression '-' NUMBER { $$ = $1 - $3; }
```

```
| expression '*' expression { $$ = $1 * $3; }
| expression '/' expression { $$ = $1 / $3; }
| '-' expression { $$ = - $2; }
| '(' expression ')' rest_of_expression { $$ = $2; }
| NUMBER { $$ = $1; }
;
```

```
rest_of_expression: '+' expression { $$ = $2; }
| '-' expression { $$ = -$2; }
| '*' expression { $$ = $2; }
| '/' expression { $$ = $2; }
| /* Empty production for no additional operator */
;
%%
```

```
int main(){ printf("\nEnter Expression:
"); yyparse(); return 0;
}
```

```
void yyerror(const char *s){
    printf("\nEntered arithmetic Expression is Invalid\n\n");
}
```

### **prog.l**

```
%{
#include "prog4.tab.h" %}

%%

[ \t]      ; // Skip whitespace
[0-9]+     { yylval = atoi(yytext); return NUMBER; }
[=]        { return '='; }
[+]        { return '+'; }
[-]        { return '-'; }
[*]        { return '*'; }
```

```
[/]      { return '/'; }
[(]      { return '('; }
[)]      { return ')'; }
.         { yyerror("Invalid character"); }
```

```
%%
```

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

**c)**

**prog.y**

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

%token PLUS
%token T
%token END
%%

start: E END { printf("Valid Expression\n"); }
      ;

E: E PLUS T {
    printf("Reduce: E -> E + T\n");
  }
| T {
    printf("Shift: E -> T\n");
  }
;

%%
```

```
#include <stdio.h> #include
<stdlib.h> int yylex(void);

int main() { printf("Shift-Reduce Parsing\n");
    yyparse(); return 0;
}

void yyerror(const char *msg) {
    fprintf(stderr, "Error: %s\n", msg);
}
```

**prog.l**

```
%{
#include "prog3.tab.h" %}

%%

[Tt]    { return T; }
\+      { return PLUS; }
\n      { return END; }
.        { /* Ignore other characters */ }

%%

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

**Output:****a)**

```
D:\Semester 5\Compiler Design\Lab>a.exe

Enter Expression: 2+8*6
= 60
```

**b)**

```
D:\Semester 5\Compiler Design\Lab>a.exe

Enter Expression: 4+4*(4+4)

^Z
= 64
```

**c)**

```
D:\Semester 5\Compiler Design\Lab>a.exe
Shift-Reduce Parsing
T+T+T+T+T
Shift: E -> T
Reduce: E -> E + T
Reduce: E -> E + T
Reduce: E -> E + T
Reduce: E -> E + T
Valid Expression
^Z
```



## Experiment – 9

### Aim:

Implement menu driven program to execute any 2 code optimization techniques on given code.

### Code:

```
#include <stdio.h>

#include <string.h>

// Function to perform Constant Folding optimization
void constantFolding(char* code) {
    char* optimizedCode = strstr(code, "5 + 3");
    if (optimizedCode != NULL) {
        strncpy(optimizedCode, "8", 1);
    }
}

// Function to perform Loop Unrolling optimization
void loopUnrolling(char* code) {
    char* optimizedCode = strstr(code, "for (int i = 0; i < 10; i++)");
    if (optimizedCode != NULL) {
        strncpy(optimizedCode, "int i = 0;\nwhile (i < 10) {", 29);
    }
}

int main() {
    char code[] = "int result = 0;\nfor (int i = 0; i < 10; i++) {\n    result += 5 + 3;\n}\n\nprintf(\"Result: %d\\n\", result);\n";
    int choice;

    while (1) {
        printf("Choose a code optimization technique:\\n");
        printf("1. Constant Folding\\n");
        printf("2. Loop Unrolling\\n");
```

```
printf("3. Quit\n");
printf("Enter your choice (1/2/3): ");
scanf("%d", &choice);
if (choice == 1) {
    constantFolding(code);
    printf("Constant Folding applied.\n");
} else if (choice == 2) {
    loopUnrolling(code);
    printf("Loop Unrolling applied.\n");
} else if (choice == 3) {
    break;
} else {
    printf("Invalid choice. Please select 1, 2, or 3.\n");
}
}
printf("Optimized code:\n%s", code);
return 0;
}
```

### Output:

```
Choose a code optimization technique:
1. Constant Folding
2. Loop Unrolling
3. Quit
Enter your choice (1/2/3): 2
Loop Unrolling applied.
Choose a code optimization technique:
1. Constant Folding
2. Loop Unrolling
3. Quit
Enter your choice (1/2/3): 1
Constant Folding applied.
Choose a code optimization technique:
1. Constant Folding
2. Loop Unrolling
3. Quit
Enter your choice (1/2/3): 3
Optimized code:
int result = 0;
int i = 0;
while (i < 10) {
```

## Experiment – 10

### Aim:

Select one block or expression from C language and generate symbol table and target code for the same.

### Code:

```
int main() {  
    int a = 5;  
    int b = 10;  
    int c;  
  
    c = a + b;  
    return 0;  
}
```

### Symbol Table Generation

A symbol table is a data structure used by a compiler or interpreter where each identifier in a source program is stored along with information associated with its declaration or appearance in the source like its type, scope level, memory, location etc.

Identifier	Type	Scope	Memory Location
main	Function	Global	-
a	int	main	Loc1
b	int	main	Loc2
c	int	main	Loc3

### Target Code Generation

Target code generation is the phase in a compiler where source code is translated into machine code or an intermediate code. Let's assume we are generating pseudo assembly code for the above snippet.

```
_main:  
    ; int a = 5;  
    MOV R1, 5    ; Move value 5 into register R1  
    STR R1, Loc1 ; Store the value from R1 into memory location Loc1 (a)
```

```
; int b = 10;
MOV R2, 10    ; Move value 10 into register R2
STR R2, Loc2  ; Store the value from R2 into memory location Loc2 (b)

; c = a + b;
LDR R3, Loc1  ; Load value from Loc1 (a) into register R3
LDR R4, Loc2  ; Load value from Loc2 (b) into register R4
ADD R3, R4    ; Add values in R3 and R4
STR R3, Loc3  ; Store result into Loc3 (c)

; return 0;
MOV R5, 0     ; Move 0 into R5 for return
RET          ; Return from function
```

pseudo assembly code is a simplified representation of what target code might look like. It demonstrates the basic operations of moving data between registers and memory and performing an addition. In a real compiler, the target code will be much more complex and optimized, and it will be specific to the target machine's architecture.