VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Compiler Design

(21CS5PCCPD)

Submitted by

Lingaraj G Mannur (1BM21CS097)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-5600

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Compiler Design**" carried out by **Lingaraj G Mannur (1BM21CS097),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester May-2023 to July-2023. The Lab report has been approved as it satisfies the academic requirements in respect of **Compiler Design (21CS5PCCPD)** work prescribed for the said degree.

Name of the Lab-In charge:

Sunayana S	Dr. Jyothi S Nayak
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Part B		
6	Write a program to implement : (a)Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A →ab /a (b)Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A → a / ab	14 - 16
Part C		
7	Use YACC to implement, evaluator for arithmetic expressions (Desktop calculator)	17-19
8	Use YACC to convert: Infix expression to Postfix expression.	20-22
9	Use YACC to generate Syntax tree for a given expression	23-25
10	Use YACC to generate 3-Address code for a given expression	26-28

Course Outcome

CO1	Apply the fundamental concepts for the various phases of compiler design.
CO2	Analyze the syntax and semantic concepts of a compiler.
CO3	Design various types of parsers and Address code generation problems are NP-Complete
CO4	Implement compiler principles, methodologies using lex, yacc tools

1.

Aim: Write a program to design Lexical Analyzer in C/C++/Java/Python Language (to recognize any five keywords, identifiers, numbers, operators and punctuations)

Code:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

int isKeyword(char *str) {
    char keywords[5][10] = {"int", "float", "if", "else", "while"}; int i;
    for (i = 0; i < 5; ++i) {
        if (strcmp(keywords[i], str) == 0)
            { return 1;
        }
    }
    return 0;
}
int isOperatorOrPunctuation(char ch)
    { char operators[] = "+-*/%=";
    char punctuations[] = "();,{}[]"; int i;
    for (i = 0; i < strlen(operators); ++i) { if
        (operators[i] == ch) {
            return 1;
        }
    }
    for (i = 0; i < strlen(punctuations); ++i) { if
        (punctuations[i] == ch) {
            return 1;
```

```
        if (isspace(input[i])) { i++;
            continue;
        }
        if (isalpha(input[i]) || input[i] == '_') { char token[50];
            int j = 0;
            token[j++] = input[i++];
            while (isalnum(input[i]) || input[i] == '_') { token[j++] = input[i++];
            }
            token[j] = '\0';
            if (isKeyword(token)) { printf("Keyword: %s\n",
                token);
            } else {
                printf("Identifier: %s\n", token);
            }
            continue;
        }
        if (isdigit(input[i])) { char
            token[50];
            int j = 0;
            token[j++] = input[i++]; while
            (isdigit(input[i])) {
                token[j++] = input[i++];
            }
            token[j] = '\0'; printf("Number: %s\n",
            token); continue;
        }
        if  (isOperatorOrPunctuation(input[i]))  { printf("Operator or
            Punctuation: %c\n", input[i++]); continue;
        } i++;
    }
}
int main() {
    char input[1000];
    printf("Enter the input string: "); fgets(input,
    sizeof(input), stdin);

    printf("Tokenizing the input:\n"); lexicalAnalyzer(input);
```

```
    return 0;
```

Output:

```
Enter the input string: int a=8;
Tokenizing the input:
Keyword: int
Identifier: a
Operator or Punctuation: =
Number: 8
Operator or Punctuation: ;
```

2.

Aim: Write a program in LEX to recognize Floating Point Numbers

Code:

```
digit [0-9]
num {digit}+
snum [-
+]?{num}
%{


({snum}[.]{num})|([.]{num})|({snum}[.])|([+-][.]{num}) {printf ("\n==>%s is a
floating-point number \n", yytext);                    }

({snum})        {    printf ("\n==>%s is not a floating-point number \n", yytext);
 }


%%

int yywrap( )
{
    return 1;
}

int main ()
```

Output:

```
Enter any number
-0.9

==>-0.9 is a floating-point number
```

3.

Aim: Write a program in LEX to recognize different tokens: Keywords, Identifiers, Constants, Operators and Punctuation symbols

Code:

```
%{
    #include<stdio.h> int
    flag=0;
%}

%%
int|for|while|float|double|do|char {  printf(" Keyword:%s\n",yytext);}
=|>=|==|<= {  printf(" Operator:%s\n",yytext);} [0-
9]* { printf(" Number:%s\n",yytext);}
[_a-zA-Z0-9|a-zA-z0-9|a-z|A-Z]* {  printf(" Identifiers:%s\n",yytext);};
;|, {  printf(" Punctuations:%s\n",yytext);}
. {}
\n  { exit(0); }
%%

int yywrap( )
{
    return 1;
```

Output:

```
Enter the sentence:
int a=8;
 Keyword:int
 Identifiers:a
 Operator:=
 Number:8
 Punctuations:;
```

4.

Aim: Write a LEX program that copies a file, replacing each nonempty sequence of white spaces by a single blank

Code:

```
s[ ]

%%

[ ]([ ])* {
        fprintf(yyout," ");
    }

([ ])*(\n)([ ])* {
spaces */
        fprintf(yyout," ");
    }



%%

int main()
```

Output:

Input.txt:

input.txt

```
Hello,   Friends
Service         to humanity
is
service to          divinity.
If
  you
     don't
         know
           how
             compiler works,
then
  you don't
know how

```

Output.txt:

output.txt

```
Hello, Friends Service to humanity is service to divinity.
If you don't know how compiler works, then you don't know how
```

5.

Aim: Write a LEX program to recognize the following tokens over the alphabets {0,1,..,9} :

a) The set of all string ending in 00.

b) The set of all strings with three consecutive 222's.

c) The set of all string such that every block of five consecutive symbols contains at least two 5's.

d) The set of all strings beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to zero modulo 5.

e) The set of all strings such that the 10^{th} symbol from the right end is 1.

f) The set of all four digits numbers whose sum is 9.

g) The set of all four digital numbers, whose individual digits are in ascending order from left to right.

Code:

```
d[0-9]
%{
#include<stdio.h>
%}

%%

({d})*00 {
    printf("%s rule A\n", yytext);
}


({d})*222({d})* {
    printf("%s rule B \n", yytext);
}


(1(0)*(11|01)(01*01|00*10(0)*(11|1))*0)(1|10(0)*(11|01)(01*01|00*10(0)*(11|1))*1
0
)*  {
```

```
{d}{4} {
        int sum = 0, i;
        for(i = 0; i < 4; i++) {
                sum = sum + yytext[i] - 48;
        }
        if(sum == 9) {
                printf("%s rule F \n", yytext);
        } else {
                sum = 1;
                for(i = 0; i < 3; i++) { if(yytext[i] > yytext[i +
                        1]) {
                                sum = 0;
                                break;
                        }
                }
                if(sum == 1) {
                        printf("%s rule G \n", yytext);
                } else {
                        printf("%s doesn't match any rule\n", yytext);
                }
        }
}

({d})* {
        int i, c = 0; if(yyleng < 5) {
                printf("%s doesn't match any rule\n", yytext);
        } else {
                for(i = 0; i < 5; i++) { if(yytext[i]
                        == '5') {
                                c++;
                        }
                }
                if(c >= 2) {
                        for(; i < yyleng; i++) { if(yytext[i - 5] ==
                                '5') {
                                        c--;
                                }
                                if(yytext[i] == '5') { c++;
                                }
                                if(c < 2) {
                                        printf("%s doesn't match any rule\n", yytext); break;
                                }
```

```
            }
            if(yyleng == i) {
                printf("%s rule C\n", yytext);
            }
        } else {
            printf("%s doesn't match any rule\n", yytext);
        }
    }
}

. { continue; }
\n { exit(0); }

%%

int yywrap()
    { return 1;
```

Output :

```
Enter text
100
100 rule A
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
1010
1010 rule D
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
222
222 rule B
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
15501
15501 rule C
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
1000000001
1000000001 rule E
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
3033
3033 rule F
nachi@Nachiketha:~/Lex_Programs$ ./a.out
Enter text
1234
1234 rule G
```

6. Write a program to implement :

(a) Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A →ab /a

(b) Recursive Descent Parsing with back tracking (Brute Force Method). S→ cAd , A → a / ab

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

#define SUCCESS 1
#define FAILED 0

char *cursor;
char string[64];

int A()
{
    if (*cursor == 'a')
    {
        cursor++;
        if ((*cursor) == 'b')
        {
            cursor++;
            printf("%-16s A -> ab\n", cursor);
        }
        else
        {
            printf("%-16s A -> a\n", cursor);
        }
```

```
        if (*cursor == 'c')
        {
                cursor++; if
                (A())
                {
                        if (*cursor == 'd')
                        {
                                printf("%-16s S -> cAd\n", "EOF"); cursor++;
                                return SUCCESS;
                        }
                        else
                        {
                                return FAILED;
                        }
                }
                else
                {
                        return FAILED;
                }
        }
        else
        {
                return FAILED;
        }
}

int main()
{
        printf("Enter the string: "); scanf("%s",
        string);
        cursor = string; puts("");
        puts("Input                           Action");
        puts("    -------------------------------------------------------");

        if (S() && *cursor == '\0')
        {
                puts("    -------------------------------------------------------");
                puts("String is successfully parsed"); return 0;
        }
        else
        {
                puts("
                    -------------------------------------------------------
```

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

Output :

```
Enter the string: cabd

Input           Action
--------------------------------
cabd                 S -> cAd
d                    A -> ab
EOF                  S -> cAd
--------------------------------
String is successfully parsed
```

7. Use YACC to implement, evaluator for arithmetic expressions (Desktop calculator)

Code:

p.l

```
%{
#include<stdio.h>
#include<stdlib.h>
#include "y.tab.h" extern
int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext);return num;}
[\t ] ;
```

p.y

```
%{
#include<stdio.h>
#include<stdlib.h>
int yyerror(const char *s); int
yylex(void);
%}
%token num;
%left '+' '-'
%left '*' '/'
%left ')'
%left '('
%%
s:e {printf("Valid expression!\n");
      printf("Result:%d\n",$$); exit(0);
```

```
|'('e')' {$$=$2;}
|num {$$=$1;}
;
%%
void main()
{
printf("Enter an arithmetic expression:\n");
yyparse();
}
int yyerror(const char *s)
```

Output:

```
Enter an arithmetic expression:
2+4*3-1
Valid expression!
Result:13
```

8. Use YACC to convert: Infix expression to Postfix expression.

Code:

p.l

```
%{
#include "y.tab.h" extern
int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return digit;} [\t] ;
[\n] return 0;
. return yytext[0];
```

p.y

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

%token digit

%%
S: E { printf("\n\n"); };
E: E '+' T { printf("+"); }
   | E '-' T { printf("-"); }
   | T
   ;
```

```
  ;

K: '(' E ')'
  | digit { printf("%d", $1); }
  ;
%%

int main()
{
    printf("Enter infix expression: "); yyparse();
    return 0;
}
```

Output:

```
Enter infix expression: 2+3-(2^4)*3
23+24^3*-
```

9. Use YACC to generate Syntax tree for a given expression

Code:

p.l

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

%%

[0-9]+   {
            yylval = atoi(yytext);
            return diait:
.           {
            fprintf(stderr, "Unknown character: %s\n", yytext);
            return 0;
        }

%%
```

p.y

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

```
{
        char val[10]; int lc;
        int rc;
};
int ind;
struct tree_node syn_tree[100]; void
my_print_tree(int cur_ind);
int mknode(int lc, int rc, const char *val);
%}

%token digit

%%
S: E { my_print_tree($1); printf("\n"); }
;

                     E: E '+' T { $$ = mknode($1, $3, "+"); }
                         | E '-' T { $$ = mknode($1, $3, "-"); }
  | T { $$ = $1; }
  ;

 T:  T      '*' F { $$      =   mknode($1,     $3,     "*");    }
   |  T      '/' F { $$      =   mknode($1,     $3,     "/");    }
  |   F     { $$ = $1;     }
  ;

 F:  K      '^' F { $$      =   mknode($1,     $3,     "^");    }
  |   K     { $$ = $1;     }
  ;

K: '(' E ')' { $$ = $2; }
  | digit { char buf[10]; sprintf(buf, "%d", yylval); $$ = mknode(-1, -1, buf); }
  ;

%%

int main()
{
        ind = 0;
        printf("Enter an expression:\n"); yyparse();
        return 0;
}
```

```
int yyerror()
{
	printf("NITW Error\n");
	return 0;
}

int mknode(int lc, int rc, const char *val)
{
	strcpy(syn_tree[ind].val, val);
	syn_tree[ind].lc = lc; syn_tree[ind].rc
	= rc;
	ind++;
	return ind - 1;
}

void my_print_tree(int cur_ind)
{
	if (cur_ind == -1)
		return;
```

Output:

```
Enter an expression:
2+3*4-(2^1)/2
Operator Node -> Index: 10, Value: -, Left Child Index: 4, Right Child Index: 9
Operator Node -> Index: 4, Value: +, Left Child Index: 0, Right Child Index: 3
Digit Node -> Index: 0, Value: 2
Operator Node -> Index: 3, Value: *, Left Child Index: 1, Right Child Index: 2
Digit Node -> Index: 1, Value: 3
Digit Node -> Index: 2, Value: 4
Operator Node -> Index: 9, Value: /, Left Child Index: 7, Right Child Index: 8
Operator Node -> Index: 7, Value: ^, Left Child Index: 5, Right Child Index: 6
Digit Node -> Index: 5, Value: 2
Digit Node -> Index: 6, Value: 1
Digit Node -> Index: 8, Value: 2
```

9. Use YACC to generate 3-Address code for a given expression

Code:

p.l

```
d [0-9]+
a [a-zA-Z]+
%{
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"
extern int yylval;
extern char iden[20];
%}
%%
{d} { yylval=atoi(yytext); return digit; }
{a} { strcpy(iden,yytext); yylval=1; return id;} [ \t]
```

p.y

```
%{
#include <math.h>
#include<ctype.h>
#include<stdio.h>   int
yyerror(char *s); int
yylex(void);
int var_cnt=0;
char iden[20];
%}
```

```
|E '-' T { $$=var_cnt; var_cnt++; printf("t%d = t%d - t%d;\n", $$, $1, $3 );}
|T {$$=$1;}
;
T:T '*' F {$$=var_cnt; var_cnt++; printf("t%d = t%d * t%d;\n", $$, $1, $3 );}
|T '/' F {$$=var_cnt; var_cnt++; printf("t%d = t%d / t%d;\n", $$, $1, $3 );}
|F {$$=$1;}
;
F:P '^' F {$$=var_cnt; var_cnt++; printf("t%d = t%d ^ t%d;\n", $$, $1, $3 );}
|P {$$ = $1;}
;
P: '(' E ')' {$$=$2;}
|digit {$$=var_cnt; var_cnt++; printf("t%d = %d;\n",$$,$1);}
;
%%
int main()
{
var_cnt=0;
printf("Enter an expression:\n");
```

Output:

```
Enter an expression:
a=2+3-(2^3)/4+2*3
t0 = 2;
t1 = 3;
t2 = t0 + t1;
t3 = 2;
t4 = 3;
t5 = t3 ^ t4;
t6 = 4;
t7 = t5 / t6;
t8 = t2 - t7;
t9 = 2;
t10 = 3;
t11 = t9 * t10;
t12 = t8 + t11;
a=t12
```