FACULTY OF ENGINEERING & TECHNOLOGY
SANKALCHAND PATEL COLLEGE OF ENGINEERING VISNAGAR

DEPARTMENT OF COMPUTER ENGINEERING& INFORMATION TECHNOLOGY
B. Tech – SEMESTER – VII | ACEDEMIC YEAR 2020-2021

PRACTICAL LIST

SUBJETC NAME: COMPILER DESIGN (CD) **SUBJECT CODE : 1ET1030703**

==

Practical-1

1) Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
void check(char t[]);
char
key[24]
[12]={"int","float","char","double","bool","void","extern","unsigned","goto","static","cla
ss","struct","for","if","else","return","register","long","while","do","include","define"};
char header[6][15]={"stdio.h","conio.h","malloc.h","process.h","string.h","ctype.h"};
int numflag=0,f=0;
void main()
{
        char ch,token[20];
        int i,lineno=0;
        FILE *fp;
        fp=fopen("input.c","r");
        if(fp==NULL)
        {
                printf("\n file does not exist");
                exit(0);
        }
        while((ch=fgetc(fp))!=EOF)
        {
                if(ch=='/')//checking for comments in the file
                {
                ch=getc(fp);
                        if(ch=='/')
                        {
```

```
                while((ch=getc(fp))!='\n');
                lineno++;
            }
            else if(ch=='*')
            {

            while(f==0)
                {
                    ch=getc(fp);
                if(ch=='\n')
                lineno++;
            else if(ch=='*')
                    {
                    ch=getc(fp);
                        if(ch=='/')
                        f=1;
                    }
                }
                f=0;
            }
        }
        else if(isalpha(ch))
        {
            i=0;
            token[i++]=ch;
            while(isalpha(ch=fgetc(fp))||isdigit(ch)||ch=='_'||ch=='.')
            {
                token[i++]=ch;
            }
            token[i]='\0';
            check(token);
            fseek(fp,-1,1);
        }
        else if(isdigit(ch))
        {
            i=0;
            token[i++]=ch;
            while(isdigit(ch=fgetc(fp))||ch=='.')
            {
                token[i++]=ch;
            }
            token[i]='\0';
            numflag=1;
            check(token);
            fseek(fp,-1,1);
        }
        else    if(ch=='    '||ch=='\n'||ch=='\t'|ch==','||ch==';'||ch=='('||ch==')'||ch=='{'||
ch=='}'||ch=='['||ch==']'||ch=='#')
            {
```

```
                                if(ch=='\n')
                                        {
                                lineno++;
                                        }
                                        else
                                        {
                                        printf("\n Spacial Character\t%c",ch);
                                        }
                                continue;
                                        }
                        else    if(ch=='+'||ch=='-'||ch=='*'||ch=='/'||ch=='%'||ch=='='||ch=='!'||ch=='<'||
ch=='>')
                                {
                                printf("\nOperator\t%c",ch);
                                continue;
                                }
                        else
                        {
                                i=0;
                                token[i++]=ch;
                                token[i]='\0';
                        check(token);
                        }
                }
  printf("\n\n\n\t\t\t Total No Of Lines Of Program is %d\n\n\n",lineno);
}
void check(char t[])
{
int i;
if(numflag==1)
{
printf("\nConstant \t %s",t);
return;
}
for(i=0;i<6;i++)
{
if(strcmp(t,header[i])==0)
{
printf("\nHeader file\t %s",t);
return;
}
}
for(i=0;i<21;i++)
{
if(strcmp(key[i],t)==0)
{
printf("\nKeyword\t\t %s",key[i]);
return;
}
```

```
}
printf("\nIdentifier\t %s",t);
}
```

Input File: input.c

```
#include<stdio.h>
void main()
{
	//hello 123
	int hr12=45;
	/* this
				is
					multi comment
								*/
}
```

Practical-2

2) Write a C program to identify whether a given line is a comment or not.

```
#include<stdio.h>

#include<conio.h>

void main()

{

char com[30];

int i=2,a=0;

clrscr();

printf("\n Enter comment:");

gets(com);

if(com[0]=='/') {

if(com[1]=='/')

printf("\n It is a comment");

else if(com[1]=='*') {

for(i=2;i<=30;i++)

{

if(com[i]=='*'&&com[i+1]=='/')

{

printf("\n It is a comment");

a=1;

break; }

else
continue; }

if(a==0)

printf("\n It is not a comment");

}

else

printf("\n It is not a comment");

}

else

printf("\n It is not a comment");

getch(); }
```

Practical-3

3) Write a C program to test whether a given identifier is valid or not.

```
#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char a[10];

int flag, i=1;

clrscr();

printf("\n Enter an identifier:");

gets(a);

if(isalpha(a[0]))

flag=1;

else

printf("\n Not a valid identifier");

while(a[i]!='\0')

{

if(!isdigit(a[i])&&!isalpha(a[i]))

{

flag=0;

break;

}
i++;

}

if(flag==1)

printf("\n Valid identifier");

getch();
}
```

Practical-4

4) To Study about Lexical Analyzer Generator(LEX) .

- **What is Lexical Analyzer Generator (LEX)**

Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for "lexical analyzer generator." It is intended primarily for Unix-based systems. ... Lex is proprietary but versions based on the original code are available as open source.

- **How do you write a Lex program?**

  To compile a lex program, do the following:

  1. Use the lex program to change the specification file into a C language program. The resulting program is in the lex. yy. ...
  2. Use the cc command with the -ll flag to compile and link the program with a library of lex subroutines. The resulting executable program is in the a.

- **How Lex tool is used in compiler design?**

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

- **What is output of Lex compiler?**

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

- **How do I run a Lex program on Windows?**

Compilation & Execution of your Program:

1. Open Command prompt and switch to your working directory where you have stored your lex file (". l") and yacc file (". y")
2. Let your lex and yacc files be "hello. l" and "hello. y". Now, follow the preceding steps to compile and run your program

- **What are the issues of lexical analyzer?**

  Issues in Lexical Analysis

  1. Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
  2. Compiler efficiency is improved.
  3. Compiler portability is enhanced.

Practical-5

5) Implement following programs using Lex.

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

```
%{
        #include <stdio.h>
%}
%%
[ \t] ;
[0-9]+\.[0-9]+ {printf("\nFound a floating-point number:");
                        printf("%s",yytext);}
[0-9]+  {printf("\nFound an integer:"); printf("%s",yytext); }
[a-zA-Z0-9]+ {printf("\nFound a string: ");printf("%s",yytext);}
%%
main() {
    // lex through the input:
    yylex();
}
```

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

```
%%
[aeiouAEIOU] {printf("\nit is vowel");}
[a-zA-Z] {printf("\nit is consonant");}
%%
 main()
{
        yylex();
        return 0;
}
```

Practical-6

6) Implement following programs using Lex.

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

```
d [0-9]
l [a-zA-Z]
o [\n ~!@#$%&*()_=\+_{}""<>?,./;:'^-]
FILE *yyin;
%%
({l}|{o})* { ;}
%%
int main()
{
yyin=fopen("lex.txt","r");
yylex();
}
```

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

```
d [0-9]
l [a-zA-Z]
o [\n ~!@#$%&*()_+|=\;',./?"":{}^-]
%%
        "<"({l}|{d})*">" { printf("%s",yytext); }
        "</"({l}|{d})*">" { printf("%s",yytext); }
        ({l}|{d})* { ;}
        ({o}) { ;}
%%
int main()
        {
                yyin=fopen("html.txt","r");
                yylex();
        }
```

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

```
int line=0;
%%
"\n" { line++; printf("\t%d",line); printf("%s",yytext); }
%%
int main()
{
        yyin=fopen("lex.txt","r");
        yylex();
}
```

Practical-7

7) Write a Lex program to count the number of comment lines in a given C program. Also eliminate them and copy that program into separate file.

CODE: (comment.l)

```
%{#include<stdio.h>
int com=0;
%}
%s COMMENT
%%
“/*” {BEGIN COMMENT;}
<COMMENT>”*/” {BEGIN 0;com++;}
<COMMENT>\n {com++;}
<COMMENT>. {;}
\/\/.* {; com++;}
.|\n {fprintf(yyout,”%s”,yytext);}
%%
void main(int argc, char *argv[])
{
if(argc!=3)
{
printf(“usage : ./a.out in.txt out.txt\n”);
exit(0);
}
yyin=fopen(argv[1],”r”);
yyout=fopen(argv[2],”w”);
yylex();
printf(“\n number of comments are = %d\n”,com);
}
int yywrap()
{
return 1;
}
```

OUTPUT :

```
lex comment.l
cc lex.yy.c -ll
./a.out in.txt out.txt
number of comments are = 2
```

CONTENT OF **IN.TXT** FILE

```
#include<stdio.h>
int main()
{
int a,b,c; /*varible declaration*/
printf(“enter two numbers”);
scanf(“%d %d”,&a,&b);
c=a+b;//adding two numbers
printf("sum is %d",c);
return 0;
}
```

CONTENT OF OUT.TXT FILE

```
#include<stdio.h>
int main()
{
int a,b,c;
printf("enter two numbers");
scanf("%d %d",&a,&b);
c=a+b;
printf("sum is %d",c);
return 0;
}
```

Practical-8

8) Write a C program for implementing the functionalities of predictive parser for the mini language.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char prol[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();
}
```

INPUT & 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->A
     S->A
     S->A
     S->A

------------------------------------------------------------------
A
     A->Bb
     A->Bb
     A->Cd
     A->Cd
------------------------------------------------------------------
B
     B->aB
     B->@
     B->@
     B->@
------------------------------------------------------------------
C
     C->@
     C->@
     C->@
```

Practical-9

9) To Study about Yet Another Compiler-Compiler(YACC).

- **What is YACC in Compiler?**

YACC stands for Yet Another Compiler Compiler. ... YACC is a program designed to compile a LALR (1) grammar. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar and the output is a C program.
yacc is a parser generator. It takes a sequence of tokens (say, from lex) and interprets them as series of statements.

- **How does yacc parser work?**

yacc then turns the specification into a C-language function that examines the input stream. This function, called a parser, works by calling the low-level scanner. The scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens.

For Compiling YACC Program:

1. **Write lex program** in a file file. l and **yacc** in a file file. y.
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type **lex** file. l.
4. type **yacc** file. y.
5. type cc **lex**. yy. c y. tab. h -ll.
6. type ./a. out.

Practical-10

10) Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .

Lex Specification

```
..................................
%{
#include “y.tab.h”
%}
%%
([0-9]+|[0-9]*\.[0-9]+) return NUM;
. return yytext[0];
\n return 0;
%%
```

Yacc Specification

```
.......................
%{
#include
%}
%token NUM
%left ‘+’ ‘-’

%left ‘*’ ‘/’
%nonassoc UMINUS
%%
exp : exp ‘+’ exp
| exp ‘-’ exp| exp ‘/’ exp
| ‘-’ exp %prec UMINUS
| exp ‘*’ exp
| ‘(‘ exp ‘)’
| NUM
;
%%
int main()
{
printf(“\n Enter an arithmatic expression”);
yyparse();
printf(“\n Valid expression”);
return 0;
}
void yyerror(){
printf(“\n Invalid expression”);
return 0;
}
```

```
void yyerror(){
printf("\n Invalid expression");
exit(0);
}
```

Practical-11

11) Create Yacc and Lex specification files are used to generate a calculator which accepts,integer and float type arguments.

Lex Specification

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

                \n              return 0;

        .               return yytext[0];
%%
yywrap()
         { return 1; }
```

Yacc Specification

```
%{
#include <stdio.h>
%}
%token OPND
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%%
valid: exp      {printf("Result: %d\n",$1); exit(0);}
     ;
exp: exp'+'exp        {$$=$1+$3;}
     |exp'-'exp       {$$=$1-$3;}
     |exp'/'exp       {if($3==0) yyerror(); $$=$1/$3; }
     |exp'*'exp       {$$=$1*$3;}
     |'-'exp %prec UMINUS          {$$=-$2;}
     |'('exp')'       {$$=$2;}
     |OPND
     ;
%%
int yyerror()
{
     printf("Invalid Expression \n");
     exit(1);
}
int main(void)

{
     if(yyparse()==0)
             printf("Valid Expression\n");
     return 0;

}
```