****

**Department of Computer Science and Engineering**

**LAB MANUAL**

**Branch:** B.Tech- CSE

**Year & Semester:** 3 year 6sem

**PCS-601 Compiler Design**

**Vision and Mission of the Department of Computer Sc. and Engineering**

**Vision**

To impart quality education for producing world class technocrats and entrepreneurs with sound ethics, latest knowledge and innovative ideas in Computer Science and Engineering to meet industrial needs and societal expectations.

**Mission**

1. To impart world class value based technical education in all aspects of Computer Science and Engineering through state of the art infrastructure and innovative approach.
2. To produce ethical, motivated and skilled engineers through theoretical knowledge and practical applications.
3. To inculcate ability for tackling simple to complex individually as well as in a team..
4. To develop globally competent engineers with strong foundations, capable of “out of the box” thinking so as to adapt to the rapidly changing scenarios requiring socially conscious green computing solutions.

**Program Educational Objectives (PEOs)**

1. To produce the students employable towards building a successful career based on sound understanding of theoretical and applied aspects and methodology to solve multidisciplinary real life problems.
2. To produce professional graduates ready to work with a sense of responsibility, ethics and enabling them to work efficiently individually and also in team.
3. To inculcate competent students so that they are able to pursue higher studies and research in areas of engineering and other professionally related fields.
4. To inculcate ability to adapt to the changing technology through continuoual learning.

**PROGRAM SPECIFIC OUTCOMES (PSOs)**

PSO1. Ability to analyze, design, implement, and test software systems based on requirement specifications and development methodologies of software systems.

PSO2. Apply computer science theory blended with engineering mathematics to solve computational tasks and model real world problems using appropriate programming language, data structure, and algorithms.

PSO3. Ability to explore technological advancements in various domains, evaluate its merits and identify research gaps to provide solution to new ideas and innovations.

**Course Outcomes**

CO1. Understand the various phases and fundamental principles of compiler design like lexical, syntactical, semantic analysis, code generation and optimization.

CO2. Compare and contrast various parsing techniques such as SLR, CLR, LALR etc

CO3. Implement lexical analyzer and parser by using modern tools like Flex and Bison

CO4. Study knowledge of patterns, tokens & regular expressions for solving a problem in the field of data mining

CO5. Design a compiler for concise programming language

CO6. Study usage of annotated tree to design the semantic rules for different aspects of programming language

List of Programs

**LEX code using Regular Grammar (without file-handling):**

1. Design a LEX Code to count the number of lines, space, tab-meta character and rest of characters in a given Input pattern.
2. Design a LEX Code to identify and print valid Identifier of C/C++ in given Input pattern.
3. Design a LEX Code to identify and print integer and float value in given Input pattern.
4. Design a LEX Code for Tokenizing (Identify and print OPERATORS, SEPERATORS, KEYWORDS, IDENTIFERS) the following C-fragment:

int p=1,d=0,r=4;

float m=0.0, n=200.0;

while (p <= 3)

{ if(d==0)

{ m= m+n\*r+4.5; d++; }

else

{ r++; m=m+r+1000.0; }

p++; }

**LEX code using Regular Grammar (with file-handling):**

1. Design a LEX Code to count and print the number of total characters, words, white spaces in given ‘Input.txt’ file.
2. Design a LEX Code to replace white spaces of ‘Input.txt’ file by a single blank character into ‘Output.txt’ file.
3. Design a LEX Code to remove the comments from any C-Program given at run-time and store into ‘out.c’ file.
4. Design a LEX Code to extract all html tags in the given HTML file at run time and store into Text file given at run time.

**LEX code using DFA:**

1. Design a DFA in LEX Code which accepts string containing even number of ‘a’ and even number of ‘b’ over input alphabet {a, b}.
2. Design a DFA in LEX Code which accepts string containing third last element ‘a’ over input alphabet {a, b}.
3. Design a DFA in LEX Code to Identify and print Integer & Float Constants and Identifier.

**YACC/LEX code:**

1. Design YACC/LEX code to recognize valid arithmetic expression with operators +, -, \* and /.
2. Design YACC/LEX code to evaluate arithmetic expression involving operators +, -, \* and / without operator precedence grammar & with operator precedence grammar.
3. Design Desk Calculator using YACC/LEX code.

.

**Program 1**

1. **Design a LEX Code to count the number of lines, space, tab-meta character and rest of characters in a given Input pattern.**

%{

int ch=0, bl=0, ln=0, wr=0;

%}

%%

[\n] {ln++;wr++;}

[\t] {bl++;wr++;}

[" "] {bl++;wr++;}

[^\n\t] {ch++;}

END {return 0;}

%%

yywrap(){}

main(int argc, char \*argv[])

{

yylex();

printf("LINES : %d WORDS : %d CHARACTERS: %d BLANK: %d",ln,wr,ch,bl);

return 0;

}

**INPUT**

This is the worst crockery around this town.

**OUTPUT**

LINES : 1 WORDS : 8 CHARACTERS: 36 BLANK: 7

**Program 2**

**Design a LEX Code to identify and print valid Identifier of C/C++ in given Input pattern.**

%{

int count=0;

%}

op [+-\*/]

letter [a-zA-Z]

digitt [0-9]

id {letter}\*|({letter}{digitt})+

notid ({digitt}{letter})+

$ {return 0;}

%%

[\t\n]+

("int")|("float")|("char")|("case")|("default")|("if")|("for")|("printf")|("scanf") {printf("%s is a keyword\n", yytext);}

{id} {printf("%s is an identifier\n", yytext); count++;}

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

%%

yywrap(){}

main(int argc, char \*argv[])

{

yylex();

printf("TOTAL IDENTIFIERS %d",count);

return 0;

}

**INPUT**

int

float

78f

90gh

a

d

are case

default

printf

scanf

**OUTPUT**

int is an identifier

float is a keyword

78f is not an identifier

90g is not an identifier

h is an identifier

a is an identifier

d is an identifier

are is an identifier

case is a keyword

default is a keyword

printf is a keyword

scanf is a keyword

TOTAL IDENTIFIERS 5

**Program 3**

Design a LEX Code to identify and print integer and float value in given Input pattern.

Hint : %{

%}

DIGIT [0-9]

%%

{DIGIT}\* {ECHO;printf(" Integer");}

{DIGIT}\*?\.{DIGIT}\* {ECHO;printf(" Float ");}

%%

yywrap(){}

main(int argc, char \*argv[])

{

yylex();

}

**INPUT**

5

3

2

66.6

22.12

3

**OUTPUT**

5 Integer

3 Integer

2 Integer

66.6 Float

22.12 Float

3 Integer

**Program 4**

**Design a LEX Code for Tokenizing (Identify and print OPERATORS, SEPERATORS, KEYWORDS, IDENTIFERS) the following C-fragment:**

**int p=1,d=0,r=4;**

**float m=0.0, n=200.0;**

**while (p <= 3)**

**{ if(d==0)**

**{ m= m+n\*r+4.5; d++; }**

**else**

**{ r++; m=m+r+1000.0; }**

**p++; }**

**Solution**

%{

#include <stdio.h>

%}

DIGIT [0-9]

NUMBER {DIGIT}+

REAL {DIGIT}\*"."{DIGIT}+

TEXT [a-zA-Z ]+

TEXT\_NUMBERS [a-zA-Z0-9]

CONDITIONALS "if"|"else"|"else if"|"switch"|"case"

KEYWORD "break"|"continue"|"goto"|"printf"|"scanf"|"sprintf"|"sscanf"|"fopen"|"fwrite"|"fread"|"fclose"|"write"|"read"|"open"|"close"|"return"|"int"|"float"|"char"|"unsigned"|"signed"|"short"|"long"|"double"

ITERATORS "for"|"while"|"do"

PREPROCESSOR "#"|"#line"|"#undef"|"#error"|"#elif"|"#else"|"#endif"|"#if"|"#define"|"#include"|"#pragma"|"#ifndef"|"#ifdef"

DELIMITER [; :\t\n()"]

IDENTIFIER [a-zA-Z]{TEXT\_NUMBERS}\*|[a-zA-Z]{TEXT\_NUMBERS}\*[[{NUMBER}+]]

FORMAT\_SPECIFIER "%"{TEXT\_NUMBERS}+

FILE "<"{IDENTIFIER}.h">"

COMMENT "/\*"[a-zA-Z0-9 \t\n;.~!@#$%^&\*()\_+=<>?:"{}]\*"\*/"

AOPERATOR "+"|"-"|"\*"|"/"|"="

BLOCK\_BEGINS "{"

BLOCK\_ENDS "}"

UNARY "++"|"--"

LOPERATOR "&"|"|"|"&&"|"~"|"||"|">"|"<"|">="|"<="|"=="

FUNCTION {IDENTIFIER}+"("{DELIMITER}\*{TEXT}{TEXT\_NUMBERS}\*{DELIMITER}\*")"

%%

{CONDITIONALS} { printf("%s is a conditional\n", yytext); }

{ITERATORS} { printf("%s is an iterator\n", yytext); }

{DIGIT} { printf("%s is a digit\n", yytext); }

{NUMBER} { printf("%s is a number\n", yytext); }

{REAL} { printf("%s is a real number\n", yytext); }

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

{DELIMITER} { printf("%s is a delimiter\n", yytext); }

{KEYWORD} { printf("%s is a keyword\n", yytext); }

{IDENTIFIER} { printf("%s is an identifier\n", yytext); }

{COMMENT} { printf("%s is a comment\n", yytext); }

{AOPERATOR} { printf("%s is a mathematical operator\n", yytext); }

{LOPERATOR} { printf("%s is a logical operator\n", yytext); }

{BLOCK\_BEGINS} { printf("Block begins\n", yytext); }

{BLOCK\_ENDS} { printf("Block ends\n", yytext); }

{FILE} { printf("%s is a file\n", yytext); }

{UNARY} { printf("%s is a unary operator\n", yytext); }

{FUNCTION} { printf("%s is a function\n", yytext); }

{FORMAT\_SPECIFIER} {printf("%s is a format specifier\n", yytext); }

%%

yywrap(){}

int main(int argc, char \*argv[]) {

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

INPUT

int p=1,d=0,r=4;

float m=0.0, n=200.0;

while (p <= 3)

{ if(d==0)

{ m= m+n\*r+4.5; d++; }

else

{ r++; m=m+r+1000.0; }

p++; }

OUTPUT

int is a keyword

is a delimiter

p is an identifier

= is a mathematical operator

1 is a digit

,d is an identifier

= is a mathematical operator

0 is a digit

,r is an identifier

= is a mathematical operator

4 is a digit

; is a delimiter

is a delimiter

float is a keyword

is a delimiter

m is an identifier

= is a mathematical operator

0.0 is a real number

, is a delimiter

n is an identifier

= is a mathematical operator

200.0 is a real number

; is a delimiter

is a delimiter

while is an iterator

is a delimiter

( is a delimiter

p is an identifier

is a delimiter

<= is a logical operator

is a delimiter

3 is a digit

) is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

Block begins

is a delimiter

if is a conditional

( is a delimiter

d is an identifier

== is a logical operator

0 is a digit

) is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

Block begins

is a delimiter

m is an identifier

= is a mathematical operator

is a delimiter

m is an identifier

+ is a mathematical operator

n is an identifier

\* is a mathematical operator

r is an identifier

+ is a mathematical operator

4.5 is a real number

; is a delimiter

is a delimiter

d is an identifier

++ is a unary operator

; is a delimiter

is a delimiter

is a delimiter

Block ends

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

else is a conditional

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

is a delimiter

Block begins

is a delimiter

r is an identifier

++ is a unary operator

; is a delimiter

is a delimiter

m is an identifier

= is a mathematical operator

m is an identifier

+ is a mathematical operator

r is an identifier

+ is a mathematical operator

1000.0 is a real number

; is a delimiter

is a delimiter

is a delimiter

Block ends

is a delimiter

is a delimiter

is a delimiter

p is an identifier

++ is a unary operator

; is a delimiter

is a delimiter

is a delimiter

Block ends

**Program 5**

Design a LEX Code to count and print the number of total characters, words, white spaces in given ‘Input.txt’ file.

%{

int lines=0, words=0,spaces=0,total=0;

%}

%%

\n { lines++; words++; total++;}

[\t ' '] {words++;total++;}

[" "] {spaces++;total++;}

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

printf("LINES : %d WORDS : %d SPACES %d TOTAL: %d",lines,words,spaces,total);

return 0;

}

**INPUT**

This is the worst crockery around this town.

**OUTPUT**

LINES : 1 WORDS : 8 SPACES 7 TOTAL: 8

**Program 6**

**Design a LEX Code to replace white spaces of ‘Input.txt’ file by a single blank character into ‘Output.txt’ file.**

%{

%}

space [ \t]

emptyline \n

%%

{space}+ printf(" ");

{emptyline}+ printf("\n");

. ECHO;

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

**INPUT**

This is the worst crockery around this town.

**OUTPUT**

This is the worst crockery around this town.

**Program 7**

**Design a LEX Code to remove the comments from any C-Program given at run-time and store into ‘out.c’ file.**

%{

%}

%%

\/\/.\* ;

\/\\*(.\*\n)\*.\*\\*\/ ;

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

**INPUT**

int p=1,d=0,r=4;

float m=0.0, n=200.0; // hello

while (p <= 3)

{ if(d==0) //this is wrong

{ m= m+n\*r+4.5; d++; }

else

{ r++; m=m+r+1000.0; } // haha

p++; }

**OUTPUT**

int p=1,d=0,r=4;

float m=0.0, n=200.0;

while (p <= 3)

{ if(d==0)

{ m= m+n\*r+4.5; d++; }

else

{ r++; m=m+r+1000.0; }

p++; }

**Program 8**

**Design a LEX Code to extract all html tags in the given HTML file at run time and store into Text file given at run time**

%{

%}

%%

(“<“|”<\\”)[a-z|A-Z|0-9]\*”>” {printf(” “);}

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

INPUT

<html> heloo </html>

<html> whatever </html>

ZXXZ

zxZX

zxzX

OUTPUT

<html> heloo </html>

<html> whatever </html>

**Program 9**

**Design a DFA in LEX Code which accepts string containing even number of ‘a’ and even number of ‘b’ over input alphabet {a, b}**

%{

%}

reg (aa|bb)\*((ab|ba)(aa|bb)\*(ab|ba)(aa|bb)\*b)\*

%%

{reg} printf("%s Accepted",yytext);

.\* printf("%s Not Accepted",yytext);

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

**INPUT**

aa

aabb

aaab

**OUTPUT**

aa Accepted

aabb Accepted

aaab Not Accepted

**Program 10**

**Design a DFA in LEX Code which accepts string containing third last element ‘a’ over input alphabet {a, b}.**

**Program 11**

**Design a DFA in LEX Code to Identify and print Integer & Float Constants and Identifier.**

%{

#include<stdio.h>

%}

%%

[-,+]?[0-9]+ printf("Constants");

[a-zA-Z]+ printf("Identifiers");

%%

yywrap(){}

main(int argc, char \*argv[])

{

extern FILE \*yyin;

yyin = fopen(argv[1],"r");

yylex();

return 0;

}

**INPUT**

0

hl

hsd

22

2123

**OUTPUT**

Constants

Identifiers

Identifiers

Constants

Constants

**Program 12**

**Design YACC/LEX code to recognize valid arithmetic expression with operators +, -, \* and /.**

The following Yacc and Lex specification files are used to generate parser which

recognizes arithemetic expressions involving + and ­.

Yacc Specification File (ath.y)

%{

%}

%token NAME NUMBER EQUL PLUS MINUS

%%

Stmt  :  NAME EQUL exp

      |  exp

      ;

exp   :  NUMBER PLUS NUMBER

      |  NUMBER MINUS NUMBER

      |  NUMBER MINUS exp

      |  NUMBER PLUS exp

      ;

%%

void yyerror(char \* s)

{

  printf ( "%s\n", s);

}

int yywrap(){return 1;}

int main(void) {return yyparse();}

Lex Specification File (ath.lex)

%{

#include"y.tab.h"

%}

%%

[a­zA­Z\\_][a­zA­Z\\_0­9]\* return NAME;

[0­9]+ return NUMBER;

"+" return PLUS;

"­" return MINUS;

"=" return EQUL;

To create parser which accepts arithemetic expressions with +, ­ operators, run the

following commands

lex  ath.lex

yacc –dv ath.y

gcc –o ath y.tab.c  lex.yy.c  ­ly –lfl

Run the command “ath” and enter the following expressions

1+2+3­2  or 1­2­3­5+4

**Program 13**

**Design YACC/LEX code to evaluate arithmetic expression involving operators +, -, \* and / without operator precedence grammar & with operator precedence grammar.**

This Yacc specification file used to develop calculator which accepts single digit

operands. Also, here we are not using any lexical specification file. The necessary lexical

analysis program (yylex()) is written directly.

Yacc specification File (calc.y)

%{

#include<stdio.h>

#include<stdlib.h>

%}

%token PLUS MINUS MUL DIV NEWLINE RPAR LPAR

%token NUMBER

/\* grammar rules & actions section \*/

%%

/\* These two rules are for reading expressions from the keyboard \*/

lines : lines line

|

;

line : expr NEWLINE { printf("%d\n> ", $1); }

| NEWLINE { printf ("> "); }

;

/\* Grammar rules for integer expressions evaluation \*/

expr : expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 ­ $3; }

| term { $$ = $1; } /\* default action \*/

;

term : term MUL factor { $$ = $1 \* $3; }

| term DIV factor { if ($3 == 0)

                             yyerror("divide by zero");

  else

     $$ = $1 / $3;

}

| factor { $$ = $1; } /\* default action \*/

;

factor : LPAR expr RPAR { $$ = $2; }

| NUMBER                { $$ = $1; } /\* default action \*/

;

%%

yylex() {

        /\* My lexer \*/

int c;

 do {

 c=getchar();

 switch (c) {

           case '0': case '1': case '2': case '3': case '4': case '5': case '6':

   case '7': case '8': case '9':

      yylval= c ­ '0';

      return NUMBER;

   case '+':  return PLUS;

   case '­':  return MINUS;

   case '\*':  return MUL;

   case '/':  return DIV;

   case '(':  return LPAR;

   case ')':  return RPAR;

   case '\n':  return NEWLINE;

         }

   } while (c!= EOF);

return(EOF);

}

main() {

printf("> ");

yyparse();

}

To generate the calculator program (executable file), run the following commands.

yacc –dv calc.y

gcc –o calc y.tab.c –ly

**Program 14**

**Design Desk Calculator using YACC/LEX code.**

This Yacc and Lex specification programs are used to generate a calculator which is flexible than the previous one. It accepts, integer and float type arguments.

Yacc Specification File (calculator.y)

%{

#include <stdio.h>

%}

%union{ double   real; /\* real value \*/

        int   integer; /\* integer value \*/

      }

%token <real> REAL

%token <integer> INTEGER

%token PLUS MINUS TIMES DIVIDE LP RP NL

%type <real> rexpr

%type <integer> iexpr

%left PLUS MINUS

%left TIMES DIVIDE

%left UMINUS

%%

lines: /\* nothing \*/

     | lines line

     ;

line:  NL

     | iexpr NL

       { printf("%d) %d\n", lineno, $1);}

     | rexpr NL

       { printf("%d) %15.8lf\n", lineno, $1);}

     ;

iexpr: INTEGER

     | iexpr PLUS iexpr

       { $$ = $1 + $3;}

     | iexpr MINUS iexpr

       { $$ = $1 ­ $3;}

     | iexpr TIMES iexpr

       { $$ = $1 \* $3;}

     | iexpr DIVIDE iexpr

       { if($3) $$ = $1 / $3;

         else { fprintf(stderr, "divide by zero\n");

                yyerror();

      }

       }

     | MINUS iexpr %prec UMINUS

       { $$ = ­ $2;}

     | LP iexpr RP

       { $$ = $2;}

     ;

rexpr: REAL

     | rexpr PLUS rexpr

       { $$ = $1 + $3;}

     | rexpr MINUS rexpr

       { $$ = $1 ­ $3;}

     | rexpr TIMES rexpr

       { $$ = $1 \* $3;}

     | rexpr DIVIDE rexpr

       { if($3) $$ = $1 / $3;

         else { fprintf(stderr, "divide by zero\n");

                yyerror();

      }

       }

     | MINUS rexpr %prec UMINUS

       { $$ = ­ $2;}

     | LP rexpr RP

       { $$ = $2;}

     | iexpr PLUS rexpr

       { $$ = (double)$1 + $3;}

     | iexpr MINUS rexpr

       { $$ = (double)$1 ­ $3;}

     | iexpr TIMES rexpr

       { $$ = (double)$1 \* $3;}

     | iexpr DIVIDE rexpr

       { if($3) $$ = (double)$1 / $3;

         else { fprintf(stderr, "divide by zero\n");

                yyerror();

      }

       }

     | rexpr PLUS iexpr

       { $$ = $1 + (double)$3;}

     | rexpr MINUS iexpr

       { $$ = $1 ­ (double)$3;}

     | rexpr TIMES iexpr

       { $$ = $1 \* (double)$3;}

     | rexpr DIVIDE iexpr

       { if($3) $$ = $1 / (double)$3;

 else { fprintf(stderr, "divide by zero\n");

                yyerror();

      }

       }

     ;

%%

#include "lex.yy.c"

int lineno;

Lex Specification File (calculator.lex)

integer      [0­9]+

dreal        ([0­9]\*\.[0­9]+)

ereal        ([0­9]\*\.[0­9]+[Ee][+­]?[0­9]+)

real         {dreal}|{ereal}

nl           \n

%%

[ \t]        ;

{integer}    { sscanf(yytext, "%d", &yylval.integer);

               return INTEGER;

             }

{real}       { sscanf(yytext, "%lf", &yylval.real);

               return REAL;

             }

\+           { return PLUS;}

\­           { return MINUS;}

\\*           { return TIMES;}

\/           { return DIVIDE;}

\(           { return LP;}

\)           { return RP;}

{nl}         { extern int lineno; lineno++;

               return NL;

             }

.            { return yytext[0]; }

To create parser which accepts arithemetic expressions with +, ­ operators, run the

following commands

lex  calculator.lex

yacc –dv calculator.y

gcc –o calculator y.tab.c  lex.yy.c  ­ly –lfl