

# **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



## **LAB REPORT**

on

## **Compiler Design**

*Submitted by*

**VAIBHAV KIRAN P (1BM21CS233)**

*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-560019**

**Nov-2023 to Feb-2024**

**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 **VAIBHAV KIRAN P (1BM21CS233)**, 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 June-2023 to Sep-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **Compiler Design(22CS5PCCPD)** work prescribed for the said degree.

**Dr. Shyamala G**  
Assistant Professor  
Department of CSE  
BMSCE, Bengaluru

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

## Index

### **Part-A: Implementation of Lexical Analyzer, By using C/C++/Java/Python language and using LEX tool.**

<b>S. No.</b>	<b>Program Details</b>	<b>Page No.</b>
1	Write a program to design Lexical Analyzer in C/C++/Java/Python Language (to recognize any five keywords, identifiers, numbers, operators and punctuations)	1 - 2
2	Write a program in LEX to recognize Floating Point Numbers.	3 - 4
3	Write a program in LEX to recognize different tokens: Keywords, Identifiers, Constants, Operators and Punctuation symbols.	5 - 6
4	Write a LEX program that copies a file, replacing each nonempty sequence of white spaces by a single blank.	7
5	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 10th 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.	8 - 10

### **Part-B: Part-B: Implementation of Parsers (Syntax Analyzers) Using C/C++/Java/Python language)**

<b>S. No.</b>	<b>Program Details</b>	<b>Page No.</b>
1	Write a program to implement (a) Recursive Descent Parsing with back tracking (Brute Force	11 - 16

	Method). $S \rightarrow cAd$ , $A \rightarrow ab / a$ (b) Recursive Descent Parsing with back tracking (Brute Force Method). $S \rightarrow cAd$ , $A \rightarrow a / ab$	
2	2. Write a program to implement: Recursive Descent Parsing with back tracking (Brute Force Method). (a) $S \rightarrow aaSaa \mid aa$ (b) $S \rightarrow aaaSaaa \mid aa$ (c) $S \rightarrow aaaaSaaaa \mid aa$ (d) $S \rightarrow aaaSaaa \mid aSa \mid aa$	17 - 24

### Part-C: Syntax Directed Translation using YACC tool

S. No.	Program Details	Page No.
1	Write a program to design LALR parsing using YACC.	25 - 26
2	Use YACC to Convert Binary to Decimal (including fractional numbers)	27 - 29
3	Use YACC to implement, evaluator for arithmetic expressions (Desktop calculator)	30 -32
4	Use YACC to convert: Infix expression to Postfix expression.	33 - 35
5	Use YACC to generate Syntax tree for a given expression	36 - 38
6	Use YACC to generate 3-Address code for a given expression	39 – 41
7	Use YACC to generate the 3-Address code which contains Arrays.	42 - 44

**Practice Programs (Lex) – 45 - 60**

**Practice Programs (YACC) – 61 - 64**

## Course Outcome

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

## **Part-A: Implementation of Lexical Analyzer, By using C/C++/Java/Python language and using LEX tool.**

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

```
import re

def is_operator(char):
    return char in ['+', '-', '*', '/', '>', '<', '=']

def is_valid_identifier(token):
    return token[0].isalpha() and not token.isdigit()

def get_keywords():
    return ["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"]

def is_integer(token):
    try:
        int(token)
        return True
    except ValueError:
        return False

def lexical_analyzer(input_str):
    tokens = re.findall(r'[a-zA-Z_]\w*|[-+*/<>=]|[(,;)]|[0-9]+', input_str)
    print("Tokens: ")
    for token in tokens:
        if token in ['+', '-', '*', '/', '>', '<', '=']:
```

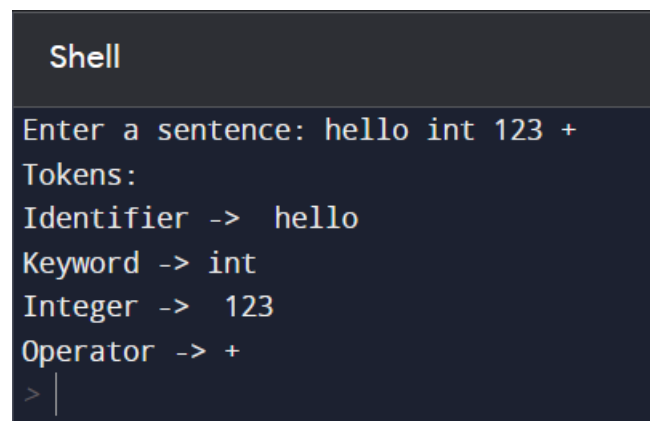
```

        print(f"Operator -> {token}")
    elif token in [' ', ';', '(', ')']:
        print(f"Delimiter -> {token}")
    elif token in get_keywords():
        print(f"Keyword -> {token}")
    elif is_integer(token):
        print(f"Integer -> {token}")
    elif is_valid_identifier(token):
        print(f"Identifier -> {token}")
    else:
        print(f"Unidentified -> {token}")

if __name__ == "__main__":
    input_string = input("Enter a C program code: ")
    lexical_analyzer(input_string)

```

### OUTPUT:



```

Shell
Enter a sentence: hello int 123 +
Tokens:
Identifier -> hello
Keyword -> int
Integer -> 123
Operator -> +
> |

```

## 2. Write a program in LEX to recognize Floating Point Numbers.

```
% {  
#include<stdio.h>  
int cnt=0;  
% }  
sign [+|-]  
num [0-9]  
dot [.]  
%%  
{sign}?{num}*{dot}{num}* {printf("Floating point no.");cnt=1;}  
{sign}?{num}* {printf("Not Floating point no.");cnt=1;}  
%%  
int yywrap()  
{  
}  
int main()  
{  
yylex();  
if(cnt==0){  
printf("Not floating pnt no.");  
}  
return 0;  
}
```



OUTPUT:

```
Enter the number: 5
Not Floating point no.
.6
Floating point no.
7.8
Floating point no.
█
```

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

```
% {  
#include<stdio.h>  
int cnt=0;  
% }  
letter [a-zA-Z]  
digit [0-9]  
punc [!|.|.]  
oper [ + | * | - | / | % ]  
boole [true|false]  
%%  
{ digit }+ | { digit } * . { digit } + { printf("Constants"); }  
int | float { printf("Keyword"); }  
{ letter } ( { digit } | { letter } ) * { printf("Identifiers"); }  
{ oper } { printf("Operator"); }  
{ punc } { printf("Punctuator"); }  
%%  
int yywrap()  
{  
}  
int main()  
{  
yylex();  
return 0;  
}
```

### OUTPUT:

```
Enter the sentence: int
Keyword
abc
Identifiers
+
Operator
!
Punctuator
123
Constants
```

**4. Write a LEX program that copies a file, replacing each nonempty sequence of white spaces by a single blank.**

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

%}

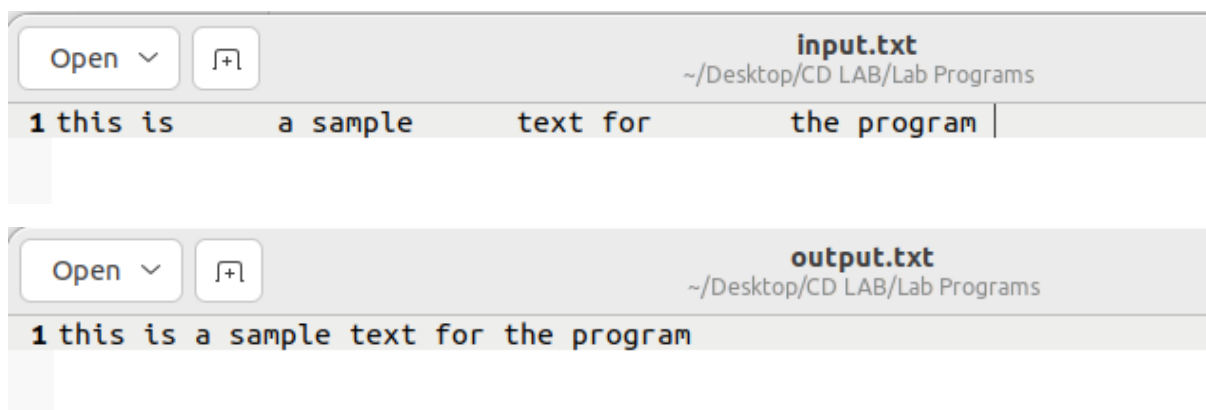
%%

[\\t" "]+ fprintf(yyout," ");
.\\n fprintf(yyout,"%s",yytext);
%%

int yywrap()
{
return 1;
}

int main(void)
{
yyin=fopen("input1.txt","r");
yyout=fopen("output.txt","w");
yylex();
return 0;
}
```

OUTPUT:



5. Write a LEX program to recognize the following tokens over the alphabets {0,1,...,9}
- The set of all string ending in 00.
  - The set of all strings with three consecutive 222's.
  - The set of all string such that every block of five consecutive symbols contains at least two 5's.
  - The set of all strings beginning with a 1 which, interpreted as the binary representation of an integer, is congruent to zero modulo 5.
  - The set of all strings such that the 10th symbol from the right end is 1.
  - The set of all four digits numbers whose sum is 9
  - The set of all four digital numbers, whose individual digits are in ascending order from left to right.

```
d[0-9]
% {
/* d is for recognising digits */
int c1=0,c2=0,c3=0,c4=0,c5=0,c6=0,c7=0;
/* c1 to c7 are counters for rules a1 to a7 */
% }
%%
({d})*00 { c1++; printf("%s -> string ending in 00\n",yytext);}
({d})*222({d})* { c2++; printf("%s -> string with three consecutive 222's \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))*10)* {
c4++;
printf("%s -> string beginning with a 1 which, interpreted as the binary representation of an
integer, is congruent to zero modulo 5 \n",yytext);
}
({d})*1{d}{9} {
c5++; printf("%s -> string such that the 10th symbol from the right end is 1 \n",yytext);
}
({d})* {
int i,c=0;
if(yyvaleng<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 -> string such that every block of five consecutive symbols contains at least two
5's\n",yytext); c3++; }
}
else
{
printf("%s doesn't match any rule\n",yytext);
}
}
}
%%

int yywrap()
{
}

int main()

```

```
{  
printf("Enter text\n");  
yylex();  
printf("Total number of tokens matching rules are : \n");  
printf("Rule A : %d \n",c1);  
printf("Rule B : %d \n",c2);  
printf("Rule C : %d \n",c3);  
printf("Rule D : %d \n",c4);  
printf("Rule E : %d \n",c5);  
return 0;  
}
```

#### OUTPUT:

```
Enter text  
1200  
1200 -> string ending in 00  
  
122299  
122299 -> string with three consecutive 222's  
  
10  
10 doesn't match any rule  
  
157495  
157495 doesn't match any rule
```

## **Part-B: Part-B: Implementation of Parsers (Syntax Analyzers) Using C/C++/Java/Python language**

**1. Write a program to implement**

**(a) Recursive Descent Parsing with back tracking (Brute Force Method).  $S \rightarrow cAd$ ,  $A \rightarrow ab/a$**

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

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

```
int S();
```

```
int A();
```

```
char input[100];
```

```
int currentIndex = 0;
```

```
int match(char symbol) {
```

```
    if (input[currentIndex] == symbol) {
```

```
        currentIndex++;
```

```
        return 1;
```

```
    } else {
```

```
        return 0;
```

```
    }
```

```
}
```

```
int S() {
```

```
    if (match('c')) {
```

```
        if (A()) {
```

```
            if (match('d')) {
```

```
                return 1;
```

```
            }
```

```
        }
```



```

    }
    return 0;
}

int A() {
    int tempIndex = currentIndex;

    if (match('a')) {
        if (match('b')) {
            return 1;
        }
    }

    currentIndex = tempIndex;

    if (match('a')) {
        return 1;
    }

    return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);

    currentIndex = 0;

    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {

```

```
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }

    return 0;
}
```

#### OUTPUT:

##### Output

```
/tmp/R63NgA7pEx.o
Enter the input string: cad
Parsing successful! Input belongs to the given grammar.
```

##### Output

```
/tmp/R63NgA7pEx.o
Enter the input string: cabd
Parsing successful! Input belongs to the given grammar.
```

##### Output

```
/tmp/R63NgA7pEx.o
Enter the input string: caab
Parsing failed! Input does not belong to the given grammar.
```

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

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

```
int S();
int A();
```

```
char input[100];
int currentIndex = 0;
```

```
int match(char symbol) {
    if (input[currentIndex] == symbol) {
        currentIndex++;
        return 1;
    } else {
        return 0;
    }
}
```

```
int S() {
    if (match('c')) {
        if (A()) {
            if (match('d')) {
                return 1;
            }
        }
    }
    return 0;
}
```

```

int A() {
    int tempIndex = currentIndex;

    if (match('a')) {
        return 1;
    }

    currentIndex = tempIndex;

    if (match('a')) {
        if (match('b')) {
            return 1;
        }
    }

    currentIndex = tempIndex;

    return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);

    currentIndex = 0;

    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }
}

```

```
    return 0;  
}
```

### OUTPUT:

#### Output

```
/tmp/R63NgA7pEx.o  
Enter the input string: cad  
Parsing successful! Input belongs to the given grammar.  
|
```

#### Output

```
/tmp/R63NgA7pEx.o  
Enter the input string: cabd  
Parsing failed! Input does not belong to the given grammar.
```

**2. Write a program to implement: Recursive Descent Parsing with back tracking (Brute Force Method).**

**(a)  $S \rightarrow aaSaa \mid aa$**

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

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

```
int S();
```

```
char input[100];
```

```
int currentIndex = 0;
```

```
int match(char symbol) {
```

```
    if (input[currentIndex] == symbol) {
```

```
        currentIndex++;
```

```
        return 1;
```

```
    } else {
```

```
        return 0;
```

```
    }
```

```
}
```

```
int S() {
```

```
    int tempIndex = currentIndex;
```

```
    if (match('a') && match('a')) {
```

```
        if (S() && match('a') && match('a')) {
```

```
            return 1;
```

```
        }
```

```
    }
```

```
    currentIndex = tempIndex;
```

```

    if (match('a') && match('a')) {
        return 1;
    }
    return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);

    currentIndex = 0;

    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }

    return 0;
}

```

### OUTPUT:

#### Output

*/tmp/R63NgA7pEx.o*

Enter the input string: aaaaaa

Parsing successful! Input belongs to the given grammar.

*/tmp/R63NgA7pEx.o*

Enter the input string: aaaa

Parsing failed! Input does not belong to the given grammar.

**(b)  $S \rightarrow \text{aaaSaaa} \mid \text{aa}$**

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

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

```
int S();
```

```
char input[100];
```

```
int currentIndex = 0;
```

```
int match(char symbol) {
```

```
    if (input[currentIndex] == symbol) {
```

```
        currentIndex++;
```

```
        return 1;
```

```
    } else {
```

```
        return 0;
```

```
    }
```

```
}
```

```
int S() {
```

```
    int tempIndex = currentIndex;
```

```
    if (match('a') && match('a') && match('a')) {
```

```
        if (S() && match('a') && match('a') && match('a')) {
```

```
            return 1;
```

```
        }
```

```
    }
```

```
    currentIndex = tempIndex;
```

```
    if (match('a') && match('a')) {
```

```
        return 1;
```

```
    }
```



```

    return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);

    currentIndex = 0;

    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }

    return 0;
}

```

#### OUTPUT:

```

Output
/tmp/eBrNhwCQKh.o
Enter the input string: aaaaaaaa
Parsing successful! Input belongs to the given grammar.

/tmp/eBrNhwCQKh.o
Enter the input string: aaaaa
Parsing failed! Input does not belong to the given grammar.

```

**(c)  $S \rightarrow \text{aaaaSaaaa} \mid \text{aa}$**

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

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

```
int S();
```

```
char input[100];
```

```
int currentIndex = 0;
```

```
int match(char symbol) {
```

```
    if (input[currentIndex] == symbol) {
```

```
        currentIndex++;
```

```
        return 1;
```

```
    } else {
```

```
        return 0;
```

```
    }
```

```
}
```

```
int S() {
```

```
    int tempIndex = currentIndex;
```

```
    if (match('a') && match('a') && match('a') && match('a')) {
```

```
        if (S() && match('a') && match('a') && match('a') && match('a')) {
```

```
            return 1;
```

```
        }
```

```
    }
```

```
    currentIndex = tempIndex;
```

```
    if (match('a') && match('a')) {
```

```
        return 1;
```

```

    }

    return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);

    currentIndex = 0;

    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }

    return 0;
}

```

### OUTPUT:

```

Output
/tmp/eBrNhwCQKh.o
Enter the input string: aaaaaaaaaa
Parsing successful! Input belongs to the given grammar.

/tmp/eBrNhwCQKh.o
Enter the input string: aaaaaaa
Parsing failed! Input does not belong to the given grammar.

```

**(d)  $S \rightarrow \text{aaaSaaa} \mid \text{aSa} \mid \text{aa}$**

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

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

```
int S();
```

```
char input[100];
```

```
int currentIndex = 0;
```

```
int match(char symbol) {
```

```
    if (input[currentIndex] == symbol) {
```

```
        currentIndex++;
```

```
        return 1;
```

```
    } else {
```

```
        return 0;
```

```
    }
```

```
}
```

```
int S() {
```

```
    int tempIndex = currentIndex;
```

```
    if (match('a') && match('a') && match('a')) {
```

```
        if (S() && match('a') && match('a') && match('a')) {
```

```
            return 1;
```

```
        }
```

```
    }
```

```
    currentIndex = tempIndex;
```

```
    if (match('a') && S() && match('a')) {
```

```
        return 1;
```

```
    }
```

```

currentIndex = tempIndex;

if (match('a') && match('a')) {
    return 1;
}

return 0;
}

int main() {
    printf("Enter the input string: ");
    scanf("%s", input);
    currentIndex = 0;
    if (S() && currentIndex == strlen(input)) {
        printf("Parsing successful! Input belongs to the given grammar.\n");
    } else {
        printf("Parsing failed! Input does not belong to the given grammar.\n");
    }

    return 0;
}

```

#### OUTPUT:

##### Output

```
/tmp/eBrNhwcQKh.o
```

```
Enter the input string: aaaaaaaaaa
```

```
Parsing failed! Input does not belong to the given grammar.
```

```
/tmp/eBrNhwcQKh.o
```

```
Enter the input string: aaaaaaaa
```

```
Parsing successful! Input belongs to the given grammar.
```

## **Part-C: Syntax Directed Translation using YACC tool**

### **1. Write a program to design LALR parsing using YACC.**

#### **Lex:**

```
% {  
    #include "y.tab.h"  
    extern int yylval;  
    % }  
    %%  
  
    //If the token is an Integer number,then return it's value.  
    [0-9]+ {yylval=atoi(yytext); return digit;}  
  
    //If the token is space or tab,then just ignore it.  
    [\t] ;  
  
    //If the token is new line,return 0.  
    [\n] return 0;  
  
    //For any other token, return the first character read since the last  
    match.  
    . return yytext[0];  
    %%
```

#### **Yacc:**

```
% {  
    #include <math.h>  
    #include<ctype.h>  
    #include<stdio.h>  
    int var_cnt=0;  
    char iden[20];  
    % }  
    %token id  
    %token digit  
    %%
```

```

S:id '=' E { printf("%s=t%d\n",iden,var_cnt-1); }

E:E '+' T { $$=var_cnt; var_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }

E: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");

yyparse();

return 0;

}

yyerror()

{

printf("error");

}

```

### OUTPUT:

```

(base) usnraju@usnraju-PC:~$ cd CompilerDesignPrograms/Set_C/C1
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ yacc -d C1.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ lex C1.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ gcc y.tab.c lex.yy.c -o C1 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ ./C1
Enter infix expression: 2+3*4
Reached

(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ ./C1
Enter infix expression: 2++
NITW Error(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C1$ █

```

## 2. Use YACC to Convert Binary to Decimal (including fractional numbers)

### Lex:

```
/* definitions */

%{

// including required header files
#include<stdio.h>
#include<stdlib.h>
#include"y.tab.h"

// declaring a external variable yylval
extern int yylval;

%}

/* rules

if 0 is matched ,make yylval to 0 and return ZERO which is
variable in Yacc program

if 1 is matched ,make yylval to 1 and return ONE which is
variable in Yacc program

if . is matched ,return POINT which is variable in Yacc program

if line change , return 0

otherwise ,ignore*/

%%

0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}
"." {return POINT;}
[ \t] {;}
\n return 0;

%%
```



### **Yacc:**

```
/* definition section*/

%{

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

//define YYSTYPE double

void yyerror(char *s);

float x = 0;

%}

// creating tokens whose values are given by lex

%token ZERO ONE POINT

// following a grammer rule which is converting binary number to
decimal number (float value)

%%

L: X POINT Y {printf("%f", $1+x);}

| X {printf("%d", $$);}

X: X B {$$=$1*2+$2;}

| B {$$=$1;}

Y: B Y {x=$1*0.5+x*0.5;}

| {;}

B:ZERO {$$=$1;}

|ONE {$$=$1;};

%%

// main function

int main()

{

printf("Enter the binary number : ");

// calling yyparse function which execute grammer rules and
lex

while(yyparse());
```

```
printf("\n");  
}  
// if any error  
void yyerror(char *s)  
{  
fprintf(stdout, "\n%s", s);  
}
```

### OUTPUT:

```
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ yacc -d C2.y  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ lex C2.l  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ gcc y.tab.c lex.yy.c -o C2 -ll  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ ./C2  
Enter the binary number : 111.011  
7.375000  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ ./C2  
Enter the binary number : 101101100  
364  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ ./C2  
Enter the binary number : 10110.1100  
22.750000  
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C2$ █
```

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

#### Lex:

```
% {  
#include<stdio.h>  
#include "y.tab.h"  
extern int yylval;  
% }  
  
%%  
[0-9]+ {  
    yylval=atoi(yytext);  
    return NUMBER;  
}  
[\t] ;  
[\n] return 0;  
. return yytext[0];  
%%  
int yywrap()  
{  
return 1;  
}
```

#### Yacc:

```
% {  
#include<stdio.h>  
int flag=0;  
% }  
  
%token NUMBER  
%left '+' '-'  
%left '*' '/' '%'  
%left '(' ')'
```

```

/* Rule Section */

%%

ArithmeticExpression: E{
    printf("\nResult=%d\n", $$);
    return 0;
};

E:E+'E' {$$=$1+$3;}
|E'-'E {$$=$1-$3;}
|E'*'E {$$=$1*$3;}
|E'/'E {$$=$1/$3;}
|E%'E {$$=$1%$3;}
|'('E)' {$$=$2;}
|NUMBER {$$=$1;}
;

%%

//driver code

void main()
{
    printf("\nEnter Any Arithmetic Expression: \n");
    yyparse();
    if(flag==0)
        printf("\nEnter arithmetic expression is Valid\n\n");
    }

void yyerror()
{
    printf("\nEnter arithmetic expression is Invalid\n\n");
    flag=1;
}

```

## OUTPUT:

```
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$ yacc -d C3.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$ lex C3.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$ gcc y.tab.c lex.yy.c -o C3 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$ ./C3
Enter an expression
2+3*4
Digit : 2
Digit : 3
Digit : 4
Multiplication Operation of 3 and 4 : 12
Addition Operation 2 and 12 : 14
Answer : 14
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$ ./C3
Enter an expression
2^3^2
Digit : 2
Digit : 3
Digit : 2
Power Operation 3 ^ 2 : 9
Power Operation 2 ^ 9 : 512
Answer : 512
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C3$
```

#### 4. Use YACC to convert: Infix expression to Postfix expression.

##### Lex:

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

##### Yacc:

```
% {  
#include <ctype.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
% }  
  
%token digit  
  
%%  
S: E { printf("\n\n"); }  
;  
  
E: E '+' T { printf("+"); }  
  | E '-' T { printf("-"); }
```

| T

;

T: T '\*' F { printf("\*"); }

| T '/' F { printf("/"); }

| F

;

F: F '^' G { printf("^"); }

| G

;

G: '(' E ')'

| digit { printf("%d", \$1); }

;

%%

int main()

{

printf("Enter infix expression: ");

yyparse();

}

yyerror()

{

printf("Error");

}

## OUTPUT:

```
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ yacc -d C4.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ lex C4.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ gcc y.tab.c lex.yy.c -o C4 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ ./C4
Enter infix expression: 2+3*4
234*+

(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ ./C4
Enter infix expression: 2+3^4*5
234^5*+

(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C4$ █
```



## 5. Use YACC to generate Syntax tree for a given expression

### Lex:

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

### **Yacc:**

```
% {  
#include <math.h>  
#include <ctype.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
struct tree_node  
{  
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,char val[10]);

% }

%token digit

%%

S:E { my_print_tree($1); }

;

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

|T { $$=$1; }

;

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

|F { $$=$1 ; }

;

F:('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");

}

int mknode(int lc,int rc,char val[10])

{

strcpy(syn_tree[ind].val,val);

syn_tree[ind].lc = lc;

syn_tree[ind].rc = rc;

```

```

ind++;
return ind-1;
}

/*my_print_tree function to print the syntax tree in DLR fashion*/
void my_print_tree(int cur_ind)
{
if(cur_ind==-1) return;
if(syn_tree[cur_ind].lc==-1&&syn_tree[cur_ind].rc==-1)
printf("Digit Node -> Index : %d, Value : %s\n",cur_ind,syn_tree[cur_ind].val); else
printf("Operator Node -> Index : %d, Value : %s, Left Child Index : %d,Right Child Index :
%d \n",cur_ind,syn_tree[cur_ind].val, syn_tree[cur_ind].lc,syn_tree[cur_ind].rc);
my_print_tree(syn_tree[cur_ind].lc);
my_print_tree(syn_tree[cur_ind].rc);
}

```

### OUTPUT:

```

(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$ yacc -d C5.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$ lex C5.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$ gcc y.tab.c lex.yy.c -o C5 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$ ./C5
Enter an expression
2+3*4
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
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$ ./C5
Enter an expression
2+3+(4*5)-6
Operator Node -> Index : 8, Value : -, Left Child Index : 6, Right Child Index : 7
Operator Node -> Index : 6, Value : +, Left Child Index : 2, Right Child Index : 5
Operator Node -> Index : 2, Value : +, Left Child Index : 0, Right Child Index : 1
Digit Node -> Index : 0, Value : 2
Digit Node -> Index : 1, Value : 3
Operator Node -> Index : 5, Value : *, Left Child Index : 3, Right Child Index : 4
Digit Node -> Index : 3, Value : 4
Digit Node -> Index : 4, Value : 5
Digit Node -> Index : 7, Value : 6
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C5$

```

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

### Lex:

```
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] {;}
\n return 0;
. return yytext[0];
%%
int yywrap()
{
}
```

### Yacc:

```
%{
#include <math.h>
#include<ctype.h>
#include<stdio.h>
int var_cnt=0;
char iden[20];
%}
%token id
%token digit
```

```

%%

S:id '=' E { printf("%s=t%d\n",iden,var_cnt-1); }

E:E '+' T { $$=var_cnt; var_cnt++; printf("t%d = t%d + t%d;\n", $$, $1, $3 ); }

|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");

yyparse();

return 0;

}

yyerror()

{

printf("error");

}

```

## OUTPUT:

```
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$ yacc -d C6.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$ lex C6.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$ gcc y.tab.c lex.yy.c -o C6 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$ ./C6
Enter an expression :
result=2+3*4
t0 = 2;
t1 = 3;
t2 = 4;
t3 = t1 * t2;
t4 = t0 + t3;
result = t4
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$ ./C6
Enter an expression :
result=((2^3^1^3)*(2*3^1^3))*(2^2*2)+2+2
t0 = 2;
t1 = 3;
t2 = 1;
t3 = 3;
t4 = t2 ^ t3;
t5 = t1 ^ t4;
t6 = t0 ^ t5;
t7 = 2;
t8 = 3;
t9 = 1;
t10 = 3;
t11 = t9 ^ t10;
t12 = t8 ^ t11;
t13 = t7 * t12;
t14 = t6 * t13;
t15 = 2;
t16 = 2;
t17 = t15 ^ t16;
t18 = 2;
t19 = t17 * t18;
t20 = t14 * t19;
t21 = 2;
t22 = t20 + t21;
t23 = 2;
t24 = t22 + t23;
result = t24
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C6$
```

## 7. Use YACC to generate the 3-Address code which contains Arrays.

### Lex:

```
% {  
#include "y.tab.h"  
  
#include <stdlib.h>  
% }  
d[0-9]  
c[a-z]  
extern char yylval;  
/*
```

#### Rules:

If an alphabet from a to z is matched, it is sent as a token.

If a tab character is encountered, nothing is done.

If a new line character is encountered, code stops running.

For anything else, the first character of the matched word is sent as token.

```
*/  
%%  
{c} { yylval=yytext[0]; return(id); }  
[t] ;  
[n] return 0;  
. return yytext[0];  
%%
```

### Yacc:

```
/* definitions */  
% {  
// including required header files  
#include<stdio.h>  
#include<stdlib.h>
```

```

#include"y.tab.h"

// declaring a external variable yylval
extern int yylval;

% }

/* rules

if 0 is matched ,make yylval to 0 and return ZERO which is
variable in Yacc program
if 1 is matched ,make yylval to 1 and return ONE which is
variable in Yacc program
if . is matched ,return POINT which is variable in Yacc program
if line change , return 0
otherwise ,ignore*/

%%

0 {yylval=0;return ZERO;}
1 {yylval=1;return ONE;}
"." {return POINT;}

[ \t] {;}

\n return 0;

%%

```



## OUTPUT:

```
C7.l C7.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C7$ yacc -d C7.y
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C7$ lex C7.l
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C7$ g++ y.tab.c lex.yy.c -o C7 -ll
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C7$ ./C7
Enter size of data type :
5
Enter no of arrays :
3
Enter no of dimension of 1 array :
2
Enter dimensions of 1 array :
3 4
Enter no of dimension of 2 array :
1
Enter dimensions of 2 array :
5
Enter no of dimension of 3 array :
3
Enter dimensions of 3 array :
6 7 8
Enter Expression ending with Semicolon
x=a+b+c+e[i][j]+d[k]+f[l][m][n];
After reduction number 1
After reduction number 2
After reduction number 3
t1 = a + b
After reduction number 4
After reduction number 5
t2 = t1 + c
After reduction number 6
After reduction number 7
t3 = i * 20
After reduction number 8
After reduction number 9
t4 = j * 5
t5 = t3 + t4
After reduction number 10
t6 = e[t5]
After reduction number 11
t7 = t2 + t6
After reduction number 12
After reduction number 13
t8 = k * 5
After reduction number 14
t9 = d[t8]
After reduction number 15
t10 = t7 + t9
After reduction number 16
After reduction number 17
t11 = l * 40
After reduction number 18
After reduction number 19
t12 = m * 35
t13 = t11 + t12
After reduction number 20
After reduction number 21
t14 = n * 5
t16 = t13 + t14
After reduction number 22
t16 = n[t17]
After reduction number 23
t17 = t10 + t16
After reduction number 24
x = t17
(base) usnraju@usnraju-PC:~/CompilerDesignPrograms/Set_C/C7$
```

## PRACTICE PROGRAMS (Lex)


### WEEK 1

#### 1. Lex program to check entered character is either number or operator

```
%option noyywrap
%{
#include<stdio.h>
%}
%%
[0-9]+ {printf("number:%s\n",yytext);}
[+-] {printf("operator:%s\n",yytext);}
[ \t\n] { /*ignore whitespaces and newline*/}
[a-zA-Z]* {printf("invalid character:%s\n",yytext);}
%%

int main()
{
printf("enter");
yylex();
return 0;
}
```

#### OUTPUT:



```
Enter: int
int->keyword
;
;->separator
123zb
123zb->identifier
```

#### 2. Lex program to count the number of words in the sentence

```
%{
#include<stdio.h>
int c=0;
%}
%%
[a-zA-Z0-9]+ {c++;}
\n {printf("the count is %d",c);}
%%
int yywrap()
{
}
```

```

int main()
{
printf("enter the sentence");
yylex();
return 0;
}

```

OUTPUT:

```

Enter the sentence: hello there
The word count is 2

```

### 3. Lex program to count vowels and consonants in a sentence

```

% {
#include<stdio.h>
int vow_count=0;
int const_count=0;
% }
%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
\n {printf("vow_count=%d,const_count=%d",vow_count,const_count);}
%%
int yywrap()
{
}
int main()
{
printf("enter the string of vowels and consonants:");
yylex();
return 0;
}

```

OUTPUT:

```

Enter the string: hello
vow_count=2,const_count=3

```

### 4. Lex program to check the type of entered word

```

%option noyywrap
%{
#include<stdio.h>
%}
%%
int|char|float {printf("\n%s->keyword",yytext);}
,|; {printf("\n %s->separator",yytext);}
[a-zA-Z0-9]* {printf("\n %s->identifier",yytext);}
%%
int wrap()
{
}
int main()
{
printf("enter");
yylex();
return 0;
}

```

#### OUTPUT:

```

Enter: 18
Entered input is a number: 18
-
Entered input is a operator: -
abc
invalid character:abc

```

#### **5. Lex program to print the input as it is**

```

%%
. ECHO;
%%
int yywrap(void)
{
}
int main(void)
{
yylex();
return 0;
}

```

OUTPUT:


```
Hello  
Hello  
█
```

## WEEK 2

### 1. Write a lex program to check whether input is digit or not

```
% {
#include<stdio.h>
#include<stdlib.h>
% }
%%
^[0-9]* printf("digit");
^[^0-9][0-9]*[a-zA-Z] printf("not a digit");
.;
%%
int yywrap()
{
}
int main()
{
yylex();
return 0;
}
```

OUTPUT:



```
123
digit
abc
not a digitbc

```

### 2. Write a lex program to check whether the given number is even or odd.

```
% {
#include<stdio.h>
int i;
% }

%%

[0-9]+ {i=atoi(yytext);
        if(i%2==0)
            printf("Even");
        else
            printf("Odd");}
%%

int yywrap(){}
```

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

**OUTPUT:**

```
Enter the number: 2
Even
3
Odd
10
Even
```

### 3. Write a lex program to check whether a number is Prime or not.

```
% {
    #include<stdio.h>
    #include<stdlib.h>
    int flag,c,j;
% }

%%
[0-9]+ {c=atoi(yytext);
    if(c==2)
    {
        printf("\n Prime number");
    }
    else if(c==0 || c==1)
    {
        printf("\n Not a Prime number");
    }
    else
    {
        for(j=2;j<c;j++)
        {
            if(c%j==0)
                flag=1;
        }
        if(flag==1)
            printf("\n Not a prime number");
        else if(flag==0)
            printf("\n Prime number");
    }
}
```

```

    }
%%
int yywrap()
{
}

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

```

OUTPUT:

```

Enter a number: 13
Prime number

5
Prime number

10
Not a prime number

```

- 4. Write a lex program to recognize**
- a) identifiers
  - b) keyword-int and float
  - c) anything else as invalid tokens.

```

% {

    #include<stdio.h>
% }
alpha[a-zA-Z]
digit[0-9]
%%
(float|int) {printf("\nkeyword");}
{ alpha } ( { digit } | { alpha } ) * {printf("\nidentifier");}
{ digit } ( { digit } | { alpha } ) * {printf("\ninvalid token");}
%%
int yywrap()
{
}
int main()
{
    yylex();
    return 0;
}

```

OUTPUT:



```
int
keyword

abc123
identifier

34
invalid token
```

5. Write a lex program to identify
- a) identifiers
  - b) keyword-int and float
  - c) anything else as invalid tokens

Read these from a text file.

```
% {

    #include<stdio.h>
    char fname[25];
% }
alpha[a-zA-Z]
digit[0-9]
%%
(float|int) {printf("\nkeyword");}
{alpha}({digit}|{alpha})* {printf("\nidentifier");}
{digit}({digit}|{alpha})* {printf("\ninvalid token");}
%%
int yywrap()
{
}
int main()
{
printf("enter filename");
scanf("%s",fname);
yyin=fopen(fname,"r");
yylex();
return 0;
fclose(yyin);
}
```

OUTPUT:

```
enter filename: input.txt

keyword
identifier
```

**6. Write a Program to print invalid string if a Alpha-Numeric string is entered as input.**

```
% {  
#include<stdio.h>  
% }  
alpha [a-zA-Z0-9]*  
%%  
[0-9]* {printf("%s IS DIGIT",yytext);}  
[a-zA-Z]* {printf("\n%s is character",yytext);}  
{alpha} {printf("invalid string");}  
%%  
int yywrap()  
{  
}  
int main()  
{  
printf("enter input");  
yylex();  
return 0;  
}
```

OUTPUT:

```
Enter input: abc  
abc is character  
  
123  
123 IS DIGIT  
  
abc123  
invalid string
```

### WEEK 3

1. Lex program to count the number of comment lines (multi line comments or single line) in a program. Read the input from a file called input.txt and print the count in a file called output.txt

```
% {
#include <stdio.h>
int cc=0;
% }
%x CMNT
%%
"/*" {BEGIN CMNT;}
<CMNT>. ;
<CMNT>"*/" {BEGIN 0; cc++;}
%%
int yywrap() { }
int main(int argc, char *argv[])
{
if(argc!=3)
{
printf("Usage : %s <scr_file> <dest_file>\n",argv[0]);
return 0;
}
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
printf("\nNumber of multiline comments = %d\n",cc);
return 0;
}
```

#### OUTPUT:

```
Number of multiline comments = 1
```

## 2. Write a program in LEX to recognize Floating Point Numbers.

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

int cnt=0;
% }

sign [+|-]
num [0-9]
dot [.]

%%

{sign}?{num}*{dot}{num}* {printf("Floating point no.");cnt=1;}
{sign}?{num}* {printf("Not Floating point no.");cnt=1;}

%%

int yywrap()
{
}

int main()
{
yylex();
if(cnt==0){
printf("Not floating pnt no.");
}
return 0;
}
```

OUTPUT:

```
Enter the number: 5
Not Floating point no.
.6
Floating point no.
7.8
Floating point no.
█
```

**3. Write a program to read and check if the user entered number is signed or unsigned using appropriate meta character**

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

int cnt=0;

% }

sign [+|-]
num [0-9]
dot [.]

%%

{ sign } { num } * { dot } * { num } * { printf("Signed no.");cnt=1;}
{ num } * { dot } * { num } * { printf("Unsigned no.");cnt=1;}

%%

int yywrap()
{
}

int main()
{
yylex();
if(cnt==0){
printf("Not floating pnt no.");
}
return 0;
}
```

OUTPUT:

```
-8
Signed no.
0
Unsigned no.
7
Unsigned no.
█
```

**4. Write a program to check if the input sentence ends with any of the following punctuationmarks ( ? , fullstop , ! )**

```
% {
#include<stdio.h>
int cnt=0;
% }
punc [?|.|.!|]
chars [a-z|A-Z|0-9|" "\t]
%%
{ chars }*{ punc } {printf("Sentence ends with punc");}
{ chars }* {printf("Sentence does not end with punc");}
%%
int yywrap()
{
}
int main()
{
yylex();
return 0;
}
```

OUTPUT:

```
Hello
Sentence does not end with punctuation
Hello!
Sentence ends with punctuation
█
```

5. Write a program to read an input sentence and to check if the sentence begins with English articles (A, a, AN, An, THE and The). If the sentence starts with the article appropriate message should be printed. If the sentence does not start with the article appropriate message should be printed

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

int cnt=0;

% }

chars [a-z|A-Z|0-9]
check [A|a|AN|An|THE|The]

%%

{check}+{chars}* {printf("Begins with %s",yytext);}
{chars}* {printf("The sentence does not begins with articles");}

%%

int yywrap()
{
}

int main()
{
printf("Enter the sentence : ");}

yylex();

return 0;

}
```

OUTPUT:

```
Enter the sentence : an apple
The sentence begins with the article an
  The sentence begins with the article apple

it is apple
The sentence does not begins with articles
  The sentence does not begins with articles
  The sentence begins with the article apple
```



## PRACTICE PROGRAMS (YACC)

### WEEK 6

1. Design a suitable grammar for evaluation of arithmetic expression having + and – operators.

+ has least priority and it is left associative

- has higher priority and is right associative

#### Lex:

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

#### Yacc:

```
% {  
#include<stdio.h>  
% }  
%token NUM  
%left '+'  
%right '-'  
%%  
expr:e {printf("Valid Expression\n"); printf ("Result: %d\n",$$); return 0;}  
e:e+'e' {$$=$1+$3;}
```

```

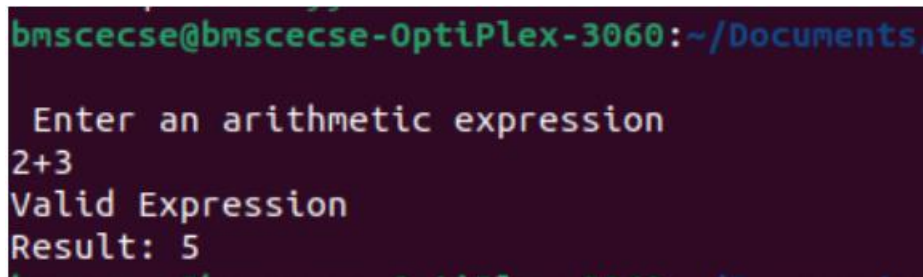
| e'-e {$$=$1-$3;}
| NUM {$$=$1;}
;
%%

int main()
{
printf("\n Enter an arithmetic expression\n");
yyparse();
return 0;
}

int yyerror()
{
printf("\nInvalid expression\n");
return 0;
}

```

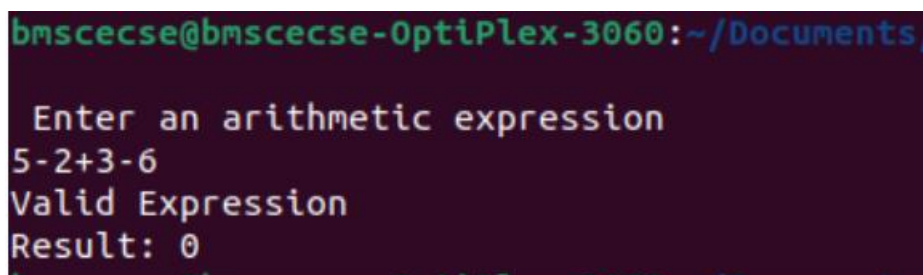
#### OUTPUT:



```

bmscecse@bmscecse-OptiPlex-3060:~/Documents
Enter an arithmetic expression
2+3
Valid Expression
Result: 5

```



```

bmscecse@bmscecse-OptiPlex-3060:~/Documents
Enter an arithmetic expression
5-2+3-6
Valid Expression
Result: 0

```

**2. Design a suitable grammar for evaluation of arithmetic expression having + , − , \* , / , % , ^ operators.**

**^ having highest priority and right associative**

**% having second highest priority and left associative**

**\*, / have third highest priority and left associative**

**+, - having least priority and left associative**

**Lex:**

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

**Yacc:**

```
% {  
#include <stdio.h>  
% }  
%token NUM  
%left '+' '-'  
%left '*' '/' '%'  
%right '^'  
%%  
expr: e { printf("Valid expression\\n"); printf("Result: %d\\n", $$); return 0; }  
e: e '+' e { $$ = $1 + $3; }
```

```

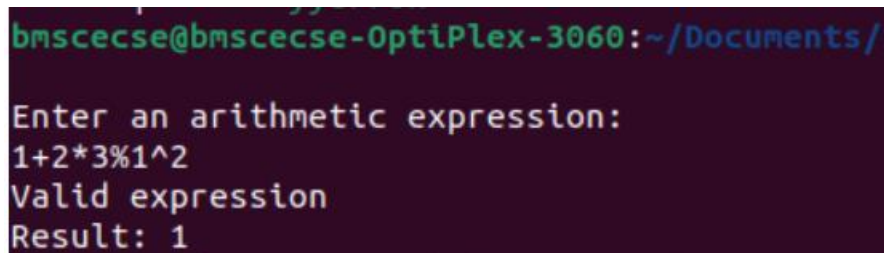
| e '-' e {$$ = $1 - $3;}
| e '*' e {$$ = $1 * $3;}
| e '/' e {$$ = $1 / $3;}
| e '%' e {$$ = $1 % $3;}
| e '^' e {
int result = 1;
for (int i = 0; i < $3; i++) {
result *= $1;
}
$$ = result;
}
| NUM {$$ = $1;}
;
%%

int main()
{
printf("\nEnter an arithmetic expression:\n");
yyparse();
return 0;
}

int yyerror()
{
printf("\nInvalid expression\n");
return 0;
}

```

#### OUTPUT:



```

bmscecse@bmscecse-OptiPlex-3060:~/Documents/
Enter an arithmetic expression:
1+2*3%1^2
Valid expression
Result: 1

```