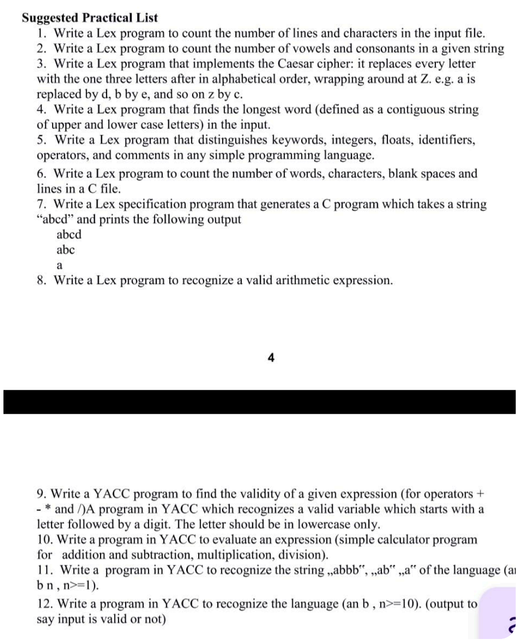

Suggested Practical List

1. Write a Lex program to count the number of lines and characters in the input file.
2. Write a Lex program to count the number of vowels and consonants in a given string
3. Write a Lex program that implements the Caesar cipher: it replaces every letter with the one three letters after in alphabetical order, wrapping around at Z. e.g. a is replaced by d, b by e, and so on z by c.
4. Write a Lex program that finds the longest word (defined as a contiguous string of upper and lower case letters) in the input.
5. Write a Lex program that distinguishes keywords, integers, floats, identifiers, operators, and comments in any simple programming language.
6. Write a Lex program to count the number of words, characters, blank spaces and lines in a C file.
7. Write a Lex specification program that generates a C program which takes a string "abcd" and prints the following output

   abcd
   abc
   a
8. Write a Lex program to recognize a valid arithmetic expression.

4

9. Write a YACC program to find the validity of a given expression (for operators + - * and /)A program in YACC which recognizes a valid variable which starts with a letter followed by a digit. The letter should be in lowercase only.
10. Write a program in YACC to evaluate an expression (simple calculator program for addition and subtraction, multiplication, division).
11. Write a program in YACC to recognize the string „abbb", „ab" „a" of the language (a
b n , n>=1).
12. Write a program in YACC to recognize the language (an b , n>=10). (output to say input is valid or not)

Please provide me a file where it is guiding me to how to each practical and also provide its code and how to run each practical
this is the compiler design subject

Thought for 2m 0s ›

Let's set you up properly for this compiler-design circus. 🧠 ⚙️

I'll give you:

- How to **run Lex & YACC programs**
- For **each practical (1–12)**:
  - What it does
  - Full **code** (Lex file `.l` and/or YACC file `.y`)
  - **Compile & run commands**

You can copy everything into files on your system.

---

0. Basic setup – how to run Lex & YACC

Your lab machine will usually have **lex / yacc** or **flex / bison** installed.

I'll write commands using the **classic** tools:

- Lex: `lex file.l` → generates `lex.yy.c`

- Yacc: `yacc -d file.y` → generates `y.tab.c` and `y.tab.h`
- Compile: `cc lex.yy.c y.tab.c -ll -ly -o progname`

If your system uses **flex/bison** instead:

- Replace `lex` → `flex`
- Replace `yacc` → `bison -d`
- Replace `-ll` → `-lfl`

I'll also assume **Linux** / WSL / Ubuntu.

PRACTICAL 1

Lex program to count number of lines and characters in the input file

File: `p1_lines_chars.l`

c Copy code

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

int lines = 0;
int chars = 0;
%}

%%
\n      { lines++; chars++; }   /* newline = one char + one line */
.       { chars++; }            /* any other single character */
%%

int main(void)
{
    yylex();    /* start scanning */
    printf("Total lines      : %d\n", lines);
    printf("Total characters : %d\n", chars);
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p1_lines_chars.l
cc lex.yy.c -ll -o p1
./p1 < input.txt
```

PRACTICAL 2

Lex program to count vowels and consonants in a given string

File: `p2_vowels_cons.l`

c Copy code

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

int vowels = 0;
int consonants = 0;
%}

%%
[aeiouAEIOU]      { vowels++; }
[A-Za-z]          { consonants++; }   /* every letter not vowel is consonant
.|\n              { /* ignore others */ }
%%

int main(void)
{
    printf("Enter a string:\n");
    yylex();

    printf("Vowels     : %d\n", vowels);
    printf("Consonants : %d\n", consonants);
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p2_vowels_cons.l
cc lex.yy.c -ll -o p2
./p2
```

Type a string, press **Enter**, then **Ctrl+D** (on Linux) to end input.

PRACTICAL 3

Lex program implementing Caesar cipher (shift by 3)

Replace each letter with the one **3 letters after** it (wrap around at `Z` and `z`).

File: `p3_caesar.l`

c Copy code

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

%%
.   {
        char c = yytext[0];
        if (c >= 'a' && c <= 'z') {
            c = ((c - 'a' + 3) % 26) + 'a';
        } else if (c >= 'A' && c <= 'Z') {
            c = ((c - 'A' + 3) % 26) + 'A';
        }
        putchar(c);   /* print transformed character */
    }
%%

int main(void)
{
    printf("Enter text to encrypt (Ctrl+D to end):\n");
    yylex();
    return 0;
}

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

Compile & run

bash Copy code

```
lex p3_caesar.l
cc lex.yy.c -ll -o p3
./p3
```

Type a sentence → Ctrl+D → encrypted version appears.

PRACTICAL 4

Lex program that finds longest word in the input

Word = contiguous sequence of **letters** (upper or lower).

File: `p4_longest_word.l`

c Copy code

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

char longest[1000] = "";
int maxlen = 0;
%}

%%
[A-Za-z]+   {
                int len = yyleng;
                if (len > maxlen) {
                    maxlen = len;
```

```c
                        strcpy(longest, yytext);
                    }
                }
.|\n            { /* ignore non-words */ }
%%

int main(void)
{
    printf("Enter text (Ctrl+D to end):\n");
    yylex();
    printf("Longest word : %s\n", longest);
    printf("Length       : %d\n", maxlen);
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p4_longest_word.l
cc lex.yy.c -ll -o p4
./p4
```

PRACTICAL 5

Lex program that distinguishes keywords, integers, floats, identifiers, operators, comments

We'll assume a **C-like tiny language**.

File: `p5_tokenizer.l`

c Copy code

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

KEYWORD   "if"|"else"|"while"|"for"|"int"|"float"|"return"|"void"
IDENT     [A-Za-z_][A-Za-z0-9_]*
INTCONST  [0-9]+
FCONST    [0-9]+"."[0-9]+
OP        [+\-*/=<>!]+
WS        [ \t]+

%%
{KEYWORD}    { printf("KEYWORD     : %s\n", yytext); }
{FCONST}     { printf("FLOAT CONST : %s\n", yytext); }
{INTCONST}   { printf("INT CONST   : %s\n", yytext); }
{IDENT}      { printf("IDENTIFIER  : %s\n", yytext); }
"//".*       { printf("COMMENT     : %s\n", yytext); }
"/*"([^*]|\*+[^*/])"*""*/"    { printf("COMMENT (block)\n"); }

{OP}         { printf("OPERATOR    : %s\n", yytext); }
{WS}         { /* skip whitespace */ }
\n           { /* ignore */ }
.            { printf("OTHER       : %s\n", yytext); }
%%

int main(void)
{
    printf("Enter code (Ctrl+D to end):\n");
    yylex();
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p5_tokenizer.l
cc lex.yy.c -ll -o p5
./p5
```

PRACTICAL 6

Lex program to count words, characters, blanks, and lines in a C file

File: `p6_wc_cfile.l`

c Copy code

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

int words = 0;
int chars = 0;
int blanks = 0;
int lines = 0;

int in_word = 0;
%}

%%
[ \t]+      { chars += yyleng; blanks += yyleng; in_word = 0; }
\n          { chars++; lines++; in_word = 0; }
[A-Za-z_0-9]+ {
                chars += yyleng;
                if (!in_word) {
                    words++;
                    in_word = 1;
                }
              }
.           { chars++; in_word = 0; }
%%

int main(void)
{
    yylex();
    printf("Lines      : %d\n", lines);
    printf("Words      : %d\n", words);
    printf("Characters : %d\n", chars);
    printf("Blanks     : %d\n", blanks);
    return 0;
}

int yywrap(void)
```

```
    {
        return 1;
    }
```

Compile & run

bash Copy code

```bash
lex p6_wc_cfile.l
cc lex.yy.c -ll -o p6
./p6 < somefile.c
```

PRACTICAL 7

Lex spec to generate program that prints

For input string `"abcd"` it should print:

text Copy code

```text
abcd
abc
ab
a
```

We'll just make a Lex program that **reads a word** and prints this pattern.

File: `p7_pattern.l`

c Copy code

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

%%
[A-Za-z]+   {
                int len = yyleng;
                while (len > 0) {
                    printf("%.*s\n", len, yytext);
                    len--;
                    }
                }
\n              { /* ignore */ }
.               { /* ignore non-letters */ }
%%

int main(void)
{
    printf("Enter a string (e.g., abcd):\n");
    yylex();
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p7_pattern.l
cc lex.yy.c -ll -o p7
./p7
```

Enter `abcd` → you get the required pattern.

PRACTICAL 8

Lex program to recognize a valid arithmetic expression

We'll handle **simple expressions** like:

- `12+3*5-7`
- `45/5+2`

No parentheses here, just numbers and + - * /.

File: `p8_valid_expr.l`

c Copy code

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

int last_type = 0;    /* 1 = number, 2 = operator */
int invalid = 0;
%}

%%
[0-9]+      {
                if (last_type == 1) {
                    /* two numbers in a row: invalid */
                    invalid = 1;
                }
                last_type = 1;
            }

[+\-*/]     {
                if (last_type != 1) {
                    /* operator cannot come first or after another operator
                    invalid = 1;
                }
                last_type = 2;
            }

[ \t\n]     { /* ignore spaces/newlines */ }

.           { invalid = 1; }  /* any other char makes it invalid */
%%

int main(void)
{
    printf("Enter expression: ");
    yylex();

    /* expression cannot end with operator */
    if (last_type != 1) invalid = 1;

    if (!invalid)
        printf("Valid arithmetic expression\n");
    else
        printf("Invalid arithmetic expression\n");
```

```c
    return 0;
}

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

Compile & run

bash Copy code

```bash
lex p8_valid_expr.l
cc lex.yy.c -ll -o p8
./p8
```

Now the **YACC** gang. For these, we usually use **both** a Lex file and a YACC file.

You can reuse almost the same Lex for 9 & 10.

Common Lex for expression YACC programs (9 & 10)

File: `expr.l`

c Copy code

```c
%{
#include "y.tab.h"
#include <stdio.h>
%}

%%
[0-9]+      {
                yylval = atoi(yytext);
                return NUMBER;
            }

[+\-*/()]   { return yytext[0]; }
```

```
\n          { return '\n'; }

[ \t]       { /* skip whitespace */ }

.           { /* ignore other chars */ }
%%

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

Compile with the corresponding YACC files below.

PRACTICAL 9

**YACC program to check validity of expression (+, -, , /)*

File: `p9_valid_expr.y`

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

int yyerror(const char *s);
int yylex(void);
%}

%token NUMBER

%%
input   : expr '\n'    { printf("Valid expression\n"); }
        | '\n'
        ;

expr    : expr '+' term
        | expr '-' term
        | term
        ;

term    : term '*' factor
```

```
            | term '/' factor
            | factor
            ;

factor  : NUMBER
            | '(' expr ')'
            ;
%%

int main(void)
{
    printf("Enter expression:\n");
    if (yyparse() == 0) {
        /* handled in grammar action */
    }
    return 0;
}

int yyerror(const char *s)
{
    printf("Invalid expression: %s\n", s);
    return 0;
}
```

Compile & run

```bash
yacc -d p9_valid_expr.y
lex expr.l
cc lex.yy.c y.tab.c -ll -ly -o p9
./p9
```

If syntax is wrong, YACC triggers `yyerror()` → prints "Invalid expression".

(Extra mini-task from your sheet)

Valid variable: lowercase letter followed by digit

If your teacher treats this as a separate YACC practical, you can use:

`var.l`

c Copy code

```c
%{
#include "y.tab.h"
%}

%%
[a-z][0-9]  { return VAR; }
\n          { return '\n'; }
.           { return yytext[0]; }
%%

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

var.y

c Copy code

```c
%{
#include <stdio.h>
int yyerror(const char *s);
int yylex(void);
%}

%token VAR

%%
input : VAR '\n'  { printf("Valid variable\n"); }
      | '\n'      { printf("Invalid variable\n"); }
      ;
%%

int main(void)
{
    printf("Enter variable (e.g., a5):\n");
    yyparse();
    return 0;
}

int yyerror(const char *s)
{
    printf("Invalid variable: %s\n", s);
    return 0;
}
```

Compile similarly with `yacc -d var.y` , `lex var.l` , etc.

PRACTICAL 10

YACC program to evaluate an expression (simple calculator)

Reuse `expr.l` from above.

File: `p10_calc.y`

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

int yyerror(const char *s);
int yylex(void);
%}

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
input   : expr '\n'    { printf("Result = %d\n", $1); }
        | '\n'
        ;

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

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

int yyerror(const char *s)
{
    printf("Error: %s\n", s);
    return 0;
}
```

Compile & run

bash Copy code

```
yacc -d p10_calc.y
lex expr.l
cc lex.yy.c y.tab.c -ll -ly -o p10
./p10
```

Input: `2+3*4` → Output: `Result = 14`

PRACTICAL 11

YACC program to recognize strings of the form $a^n b^n$ ($n \ge 1$)

Your sheet mentions strings like “abbb”, “abb”, etc. Typical language is `{ aⁿ bⁿ | n ≥ 1 }`.

Lex file: `anbn.l`

c Copy code

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

%%
"a"     { return 'a'; }
"b"     { return 'b'; }
\n      { return '\n'; }
[ \t]   { /* skip */ }
.       { return yytext[0]; }
```

```
%%

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

YACC file: `p11_anbn.y`

c Copy code

```c
%{
#include <stdio.h>
int yylex(void);
int yyerror(const char *s);
%}

%%
input   : S '\n'      { printf("Valid string of a^n b^n\n"); }
        | '\n'        { printf("Invalid string\n"); }
        ;

S       : 'a' 'b'
        | 'a' S 'b'
        ;
%%

int main(void)
{
    printf("Enter string over {a,b}:\n");
    yyparse();
    return 0;
}

int yyerror(const char *s)
{
    printf("Invalid string: %s\n", s);
    return 0;
}
```

Compile & run

bash Copy code

```bash
yacc -d p11_anbn.y
lex anbn.l
```

```
cc lex.yy.c y.tab.c -ll -ly -o p11
./p11
```

Examples:

- `ab` → valid
- `aabb` → valid
- `aaabbb` → valid
- `abb` or `aab` → invalid

PRACTICAL 12

YACC program to recognize language { a^n b | n ≥ 10 }

That means: at least **10 'a'**s followed by **single 'b'**.

Lex file: `a10b.l`

c Copy code

```c
%{
#include "y.tab.h"
%}

%%
"a"     { return 'a'; }
"b"     { return 'b'; }
\n      { return '\n'; }
[ \t]   { /* skip */ }
.       { return yytext[0]; }
%%

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

YACC file: `p12_a10b.y`

c Copy code

```c
%{
#include <stdio.h>
int yylex(void);
int yyerror(const char *s);
%}
```

```
%%
input   : S '\n'      { printf("Valid (a^n b, n>=10)\n"); }
        | '\n'        { printf("Invalid string\n"); }
        ;

S       : TEN more_a 'b'
        ;

TEN     : 'a''a''a''a''a''a''a''a''a''a'   /* exactly 10 'a's */
        ;

more_a  : more_a 'a'
        | /* empty, so n can be exactly 10 */
        ;
%%

int main(void)
{
    printf("Enter string (a^n b, n>=10):\n");
    yyparse();
    return 0;
}

int yyerror(const char *s)
{
    printf("Invalid string: %s\n", s);
    return 0;
}
```

Compile & run

bash Copy code

```bash
yacc -d p12_a10b.y
lex a10b.l
cc lex.yy.c y.tab.c -ll -ly -o p12
./p12
```

Examples:

- `aaaaaaaaaab` (10 a's + b) → valid
- `aaaaaaaaaaab` (11 a's + b) → valid

- `aaaaaaaab` (8 a's + b) → invalid
- `aaaaaaaaaabb` → invalid

That's the full "do all your practicals" kit:

- Lex only: **1–8**