MARWADI UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CLASS: 7EC3 BATCH: A

COMPILER DESIGN

(01CE0714)

2025-2026

STUDENT LAB MANUAL

Razi Ahmad **92201703004**

MARWADI UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CLASS: 7EC3 BATCH: A

INDEX

Sr. No.	Title	Date	Grade	Sign
1	Write a C Program to remove Left Recursion from grammar.			
2	Write a C Program to remove Left Factoring from grammar.			
3	Write a C program to implement finite automata and string validation.			
4	Prepare report for Lex and install Lex on Linux/Windows.			
5	(a) WALEx Program to count words, characters, lines, Vowels and consonants from given input. (b) WALEx Program to generate string which is ending with zeros.			
6	(a) WALex Program to generate Histogram of words (b) WALex Program to remove single or multi line comments from C program.			
7	WALex Program to check weather given statement is compound or simple.			
8	WALex Program to extract HTML tags from .html file.			
9	Write a C Program to compute FIRST Set of the given grammar.			
10	Write a C Program to compute FOLLOW Set of the given grammar.			
11	Write a C Program to implement Operator precedence parser.			
12	Write a C Program for constructing LL (1) parsing.			
13	Write a C program to implement SLR parsing.			
14	Prepare a report on YACC and generate Calculator Program using YACC.			

Practical 1

Title: Write a C Program to remove Left Recursion from the grammar.

Hint: Enter grammar like A->Aa|b, where the first part is left-recursive and the second is not. The program checks for immediate left recursion and rewrites the grammar without it.

Program:

prac-1.c

```
#include<stdio.h>
#define SIZE 10
void main () {
  char non_terminal;
  char beta,alpha[6];
  char production[SIZE];
  int index=3;
  int i=0,j=0;          /* starting of the string following "->" */
  printf("Enter the grammar:\n");
  scanf("%s",&production);
  non_terminal=production[0];
  if(non_terminal==production[index]) {

    for(i=index+1;production[i]!='|';i++)
    {
   alpha[j]=production[i];
   j++;
    }
    alpha[j]='\0';

    printf("Grammar is left recursive.\n");
    while(production[index]!=0 && production[index]!='|')
    index++;
    if(production[index]!=0) {
      beta=production[index+1];
      printf("Grammar without left recursion:\n");
      printf("%c->%c%c\'",non_terminal,beta,non_terminal);
      printf("\n%c\'->%s%c\'|E\n",non_terminal,alpha,non_terminal);
    } else
     printf("Grammar can't be reduced\n");
```

```
  } else
    printf("Grammar is not left recursive.\n");
}
```

Output:

Practical 2

Title: Write a C Program to remove Left Factoring from the grammar.

Hint : Left factoring is a grammar transformation technique used to make a non-deterministic grammar suitable for predictive parsing. When two or more productions for a non-terminal start with the same sequence of symbols, a parser might not be able to decide which production to choose. Left factoring involves factoring out the common prefix and creating a new non-terminal to represent the rest of the productions.

Program:

prac-2.c

```
#include <stdio.h>
#include <string.h>
int main()
{
   char production[100];
   char common[10], alpha[10], beta[10];
   char non_terminal;
   int i = 0, j = 0, k = 0;
   printf("Enter production (e.g., A->abx|aby): ");
   scanf("%s", production);
   non_terminal = production[0];
   if (production[1] != '-' || production[2] != '>')
   {
      printf("Invalid production format.\n");
      return 1;
   }
   i = 3;
   while (production[i] != '|' && production[i] != '\0')
   {
      alpha[j++] = production[i++];
   }
   alpha[j] = '\0';
   if (production[i] == '|')
      i++;
   j = 0;
   while (production[i] != '\0')
   {
```

```
        beta[j++] = production[i++];
    }
    beta[j] = '\0';
    j = 0;
    while (alpha[j] == beta[j] && alpha[j] != '\0')
    {
        common[k++] = alpha[j++];
    }
    common[k] = '\0';
    if (strlen(common) > 0)
    {
        printf("\nLeft factoring applied.\n");
        printf("The equivalent productions are:\n");
        printf("%c -> %s%c'\n", non_terminal, common, non_terminal);
        printf("%c' -> %s | %s\n", non_terminal, &alpha[j], &beta[j]);
    }
    else
    {
        printf("\nNo common prefix found. Left factoring not needed.\n");
    }
    return 0;
}
```

Output:

Practical 3

Title: Write a C program to implement finite automata and string validation.

Hint : Left factoring is a grammar transformation technique used to make a non-deterministic grammar suitable for predictive parsing. When two or more productions for a non-terminal start with the same sequence of symbols, a parser might not be able to decide which production to choose. Left factoring involves factoring out the common prefix and creating a new non-terminal to represent the rest of the productions.

Program:

prac-3.c

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

typedef enum
{
    q0,
    q1,
    q2
} State;

int isAcceptedByFA(const char *str)
{
    State currentState = q0;
    int i = 0;

    while (str[i] != '\0')
    {
        char ch = str[i];

        switch (currentState)
        {
        case q0:
            if (ch == 'a')
            {
                currentState = q0;
            }
            else if (ch == 'b')
```

```
            {
                currentState = q1;
            }
            else
            {
                currentState = q2;
            }
            break;

        case q1:
            if (ch == 'b')
            {
                currentState = q1;
            }
            else
            {
                currentState = q2;
            }
            break;

        case q2:
            // Invalid state, exit early
            return 0;
        }

        i++;
    }

    // Accept if ending state is q1
    return currentState == q1;
}

int main()
{
    char input[100];
    printf("Enter a string (only a's followed by at least one b): ");
    scanf("%s", input);

    if (isAcceptedByFA(input))
    {
```

```
        printf("The string is accepted by the finite automaton.\n");
    }
    else
    {
        printf("The string is NOT accepted by the finite automaton.\n");
    }

    return 0;
}
```

Output:

Practical 4

Title: To Study about Lexical Analyzer Generator (LEX) and Flex (Fast Lexical Analyzer).

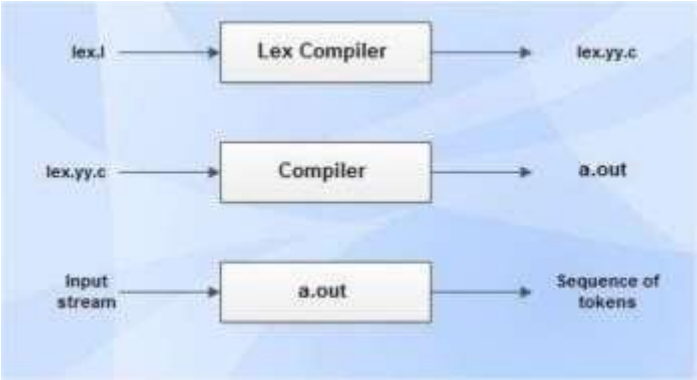
Description:

LEX: Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

Function of Lex:

1. In the first step the source code which is in the Lex language having the file name 'File.l' gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.

2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an 'a.out' file, and finally, the output file a.out will take the stream of character and generates tokens as output.



Structure of LEX Program:

%{
Definition section
%}

%%
Rules section
%%
User Subroutine section

The **Definition section** is the place to define macros and import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. It is bracketed with %{ and %).

The **Rules section** is the most important section; Each rule is made up of two parts: a pattern and an action separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. It is bracketed with %% & %%

The **User Subroutine** section in which all the required procedures are defined. It contains the main in which C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section/.

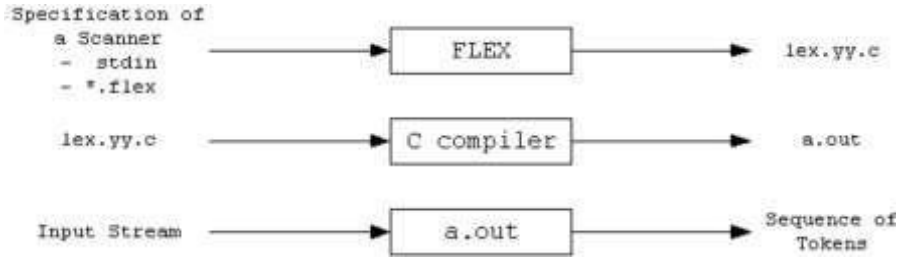
Predefined Functions and Variables of LEX:

Name	Function
`int yylex(void)`	Call to invoke lexer; returns the next token.
`char *yytext`	Pointer to the matched string.
`int yyleng`	Length of the matched string.
`yylval`	Value associated with the current token (used in parser).
`int yywrap(void)`	Called at the end of input; return 1 if done, 0 to continue scanning.
`FILE *yyout`	Output file for `ECHO` or user-defined output.
`FILE *yyin`	Input file to be scanned by the lexer.
`INITIAL`	Default (initial) start condition for the lexer.
`BEGIN condition`	Switches the scanner to a new start condition (`condition`).
`ECHO`	Writes the matched string (`yytext`) to the output (`yyout`).

Description:

FLEX: FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

Bison produces parser from the input file provided by the user. The function yylex() is automatically generated by the flex when it is provided with a .1 file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.



Structure of FLEX Program:

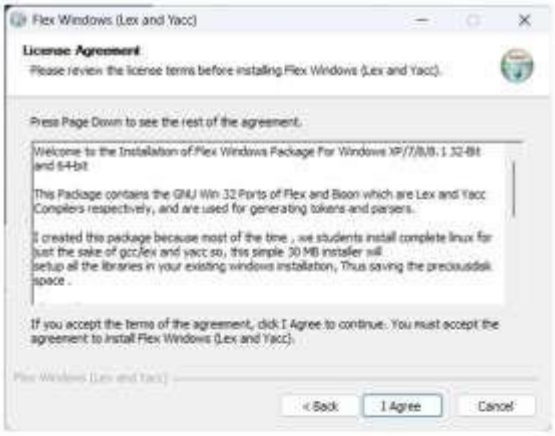
%{
// Definitions
%}
%%
Rules
%%
User code section

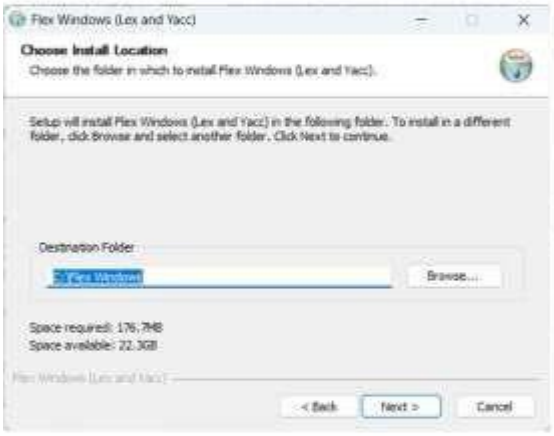
Definition Section: The definition section contains the declaration of variables, regular definitions, manifest constants. In the definition section, text is enclosed in "%{%}" brackets. Anything written in this brackets is copied directly to the file lex.yy.c

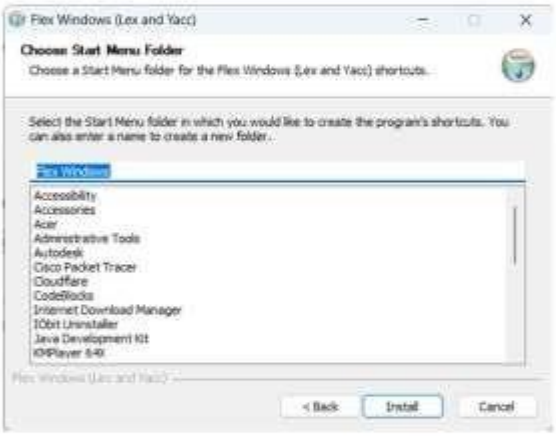
Rules Section: The rules section contains a series of rules in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in "%%%%".

User Code Section: This section contains C statements and additional functions. We can also compile these functions separately and load with the lexical analyzer.

Installation Steps of Flex







Practical 5

Title: (a) WALEx Program to count words, characters, lines, Vowels and consonants from given input.

Program:

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

int ch = 0, l = 1, w = 1, v = 0, c = 0;
%}

%%

[aeiouAEIOU] { ch++; v++; }
[a-zA-Z] { ch++; c++; }
[\t ] { w++; ch++; }
\n { l++; w++; ch++; }
. { ch++;

%%

int yywrap() { return 1; }

int main() {
   yyin = fopen("r.txt", "r");
   if (!yyin) {
      printf("Error: Cannot open input file.\n");
      return 1;
   }

   yylex(); // Start lexical analysis

   printf("No. of Characters: %d\n", ch);
   printf("No. of Lines: %d\n", l);
   printf("No. of Words: %d\n", w);
   printf("No. of Vowels: %d\n", v);
   printf("No. of Consonants: %d\n", c);

   return 0;
}
```

Output:


Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.
E:\SEM-7\COMPILER DESIGN\cd_practical>lex prac_5a.l
E:\SEM-7\COMPILER DESIGN\cd_practical>lex gcc.yy.c
lex: can't open gcc.yy.c
E:\SEM-7\COMPILER DESIGN\cd_practical>gcc lex.yy.c
E:\SEM-7\COMPILER DESIGN\cd_practical>a.exe
Enter the text:
I am Razi
Ali
Result ---
Vowels: 7
Consonants: 8
Words: 4
Characters (excluding newlines): 18
Lines: 2
E:\SEM-7\COMPILER DESIGN\cd_practical>

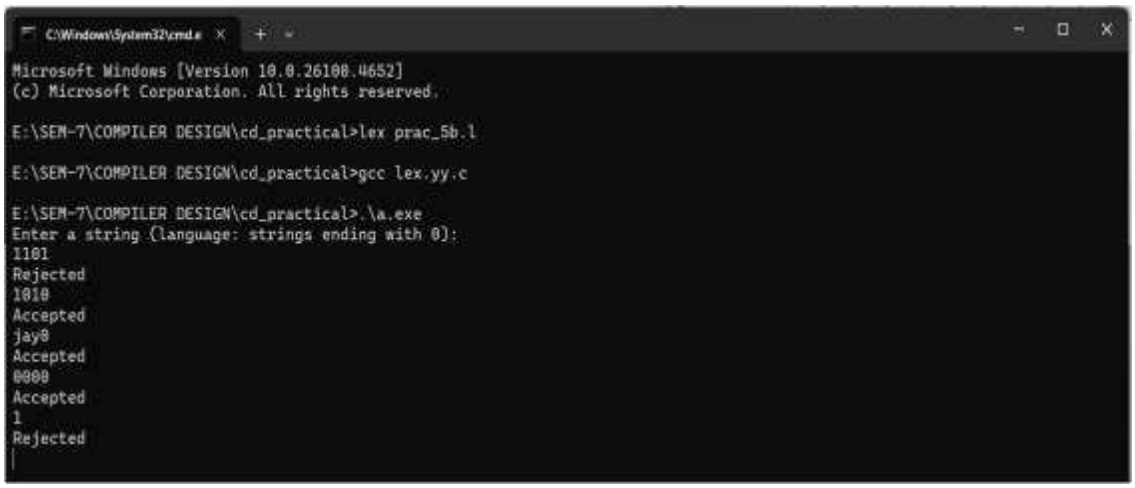
v

Title: (b) WALEx Program to generate string which is ending with zeros.

Program:

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

%%
[01]*0     { printf("Accepted\n"); }
[01]*1     { printf("Rejected\n"); }
.|\n       { /* Ignore other characters */ }
%%

int main()
{
   printf("Enter a string (language: strings ending with 0):\n");
   yylex();
   return 0;
}

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

Output:



Practical 6

Title: (a) WALex Program to generate Histogram of words.

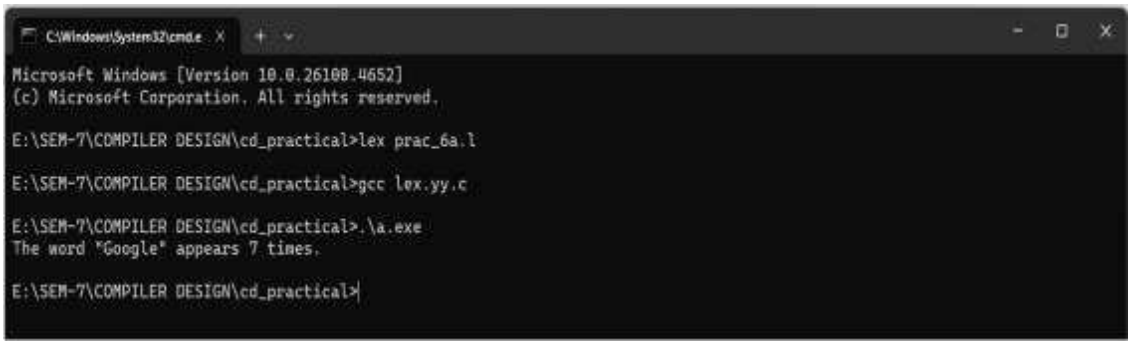
Program:

```
%{
#include <stdio.h>
#include <string.h>
char word[] = "Google";
int count = 0;
%}
%%
[a-zA-Z]+ {
   if (strcmp(yytext,word) == 0)
      count++;
}
. ;  // Ignore other characters (like punctuation, space, newline)
%%

int yywrap() {
   return 1;
}

int main() {
   yyin = fopen("r.txt", "r"); // Input text file
   yylex(); // Start lexical analysis
   printf("The word \"%s\" appears %d times.\n", word, count);
   return 0;
}
```

Text File: (File name: **r.txt)**

Google a global technology giant, has become synonymous with innovation and information. Whether you're using Google Search to find answers to your questions, exploring new places on Google Maps, or collaborating with others through Google Docs, the company's influence is undeniable. Google continues to push the boundaries of technology with its developments in artificial intelligence, cloud computing, and mobile operating systems like Android. B` integrating Google services into daily life, users worldwide benefit from the seamless and efficient tools that Google consistently provides.

Output:



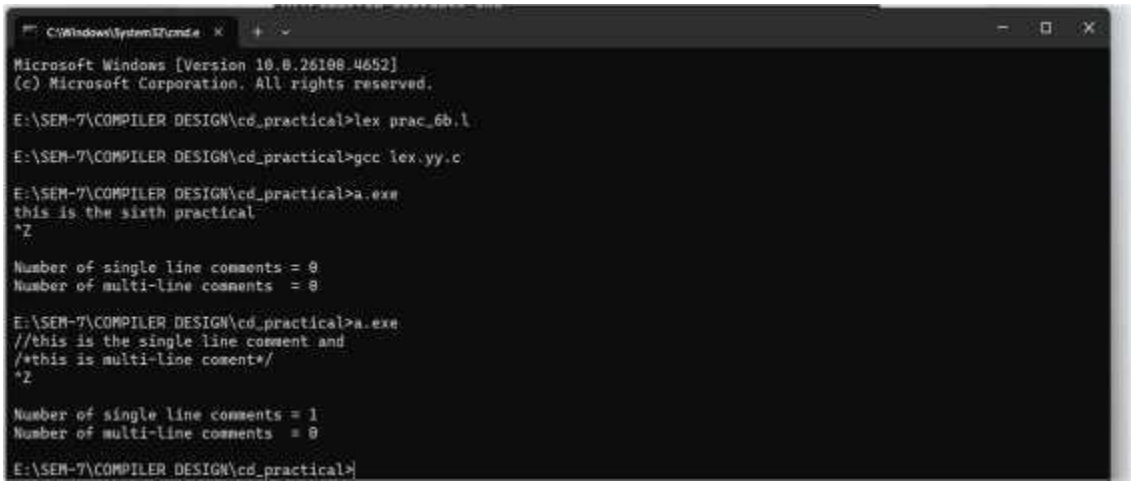
Title: (b) WALex Program to remove single or multi line comments from C program.

Hint: This Lex program reads input and counts how many times each word appears. It stores each word in an array and tracks its frequency, ignoring punctuation and whitespace.

Program:

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

int sl = 0;
int ml = 0;
%}

%%
"/*"[a-zA-Z0-9 \t\n]*"*/"      ml++;
"//".*                         sl++;
.|\n                           ;   // Ignore other characters
%%

int yywrap() {
    return 1;
}

int main() {
    yyin = fopen("f1.c", "r");
    yyout = fopen("f2.c", "w");

    yylex();

    fclose(yyin);
    fclose(yyout);

    printf("\nNumber of single line comments = %d\n", sl);
    printf("Number of multi-line comments  = %d\n", ml);

    return 0;
}
```

Output:


Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.
E:\SEM-7\COMPILER DESIGN\cd_practical>lex prac_6b.l
E:\SEM-7\COMPILER DESIGN\cd_practical>gcc lex.yy.c
E:\SEM-7\COMPILER DESIGN\cd_practical>a.exe
this is the sixth practical
^Z
Number of single line comments = 0
Number of multi-line comments = 0
E:\SEM-7\COMPILER DESIGN\cd_practical>a.exe
//this is the single line comment and
/*this is multi-line coment*/
^Z
Number of single line comments = 1
Number of multi-line comments = 0
E:\SEM-7\COMPILER DESIGN\cd_practical>