MARWADI UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CLASS: TC1 BATCH: A

COMPILER DESIGN

(01CE0714)

2025-2026

STUDENT LAB MANUAL

Yashkumar Mayani **92310103058**

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 whether 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 :

Left Recursion in grammar occurs when a non-terminal appears on the left-most side of its own production.
Example: A → Aα | β
After removing left recursion:
A → βA'
A' → αA' | ε

Program :

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

void main() {
   // prod = full input production, like A->Aa|b
   // alpha = stores recursive rules (Aa -> a), beta = non-recursive rules
   char prod[100], alpha[10][20], beta[10][20], nonTerminal;
   int i = 0, m = 0, n = 0;

   // ?? Step 1: Get user input
   printf("Enter production (e.g., A->Aa|b): ");
   scanf("%s", prod);  // Input like A->Aa|b

   // ?? Step 2: Extract non-terminal from LHS
   nonTerminal = prod[0];

   // Check format (must be like A->)
   if (!(prod[1] == '-' && prod[2] == '>')) {
      printf("Invalid production format!\n");
      return;
   }

   i = 3; // Start reading from RHS (skip A->)

   // Temporary string to hold each rule between '|'
   char temp[20];
   int t = 0;

   // ?? Step 3: Loop to split RHS rules by '|'
   while (1) {
      // End of one rule (either '|' or '\0')
      if (prod[i] == '|' || prod[i] == '\0') {
         temp[t] = '\0'; // End the string
```

```
        // If rule starts with nonTerminal ? it's left recursive
        if (temp[0] == nonTerminal)
            strcpy(alpha[m++], temp + 1);  // Remove nonTerminal and store in alpha[]
        else
            strcpy(beta[n++], temp);       // Store safe rules in beta[]

        t = 0; // Reset temp
        if (prod[i] == '\0') break; // End loop if end of string
      } else {
        temp[t++] = prod[i]; // Add character to temp rule
      }
      i++;
    }

    // ?? Step 4: If no left recursion found
    if (m == 0) {
      printf("No Left Recursion Detected.\n");
      return;
    }

    // ?? Step 5: Output new grammar after removing left recursion
    printf("After removing left recursion:\n");

    // Print: A -> beta A'
    printf("%c -> ", nonTerminal);
    for (i = 0; i < n; i++) {
      printf("%s%c'", beta[i], nonTerminal);  // Add A' to beta
      if (i != n - 1) printf(" | ");
    }

    // Print: A' -> alpha A' | e
    printf("\n%c' -> ", nonTerminal); // Proper A' (not B')
    for (i = 0; i < m; i++) {
      printf("%s%c' | ", alpha[i], nonTerminal); // alpha part + A'
    }
    printf("e\n");  // Always add epsilon (empty string) to end A' recursion
}
```

Output:

```
Enter production (e.g., A->Aa|b): A->Aab|xy
After removing left recursion:
A -> xyA'
A' -> abA' | e

--------------------------------
Process exited after 23.48 seconds with return value 0
Press any key to continue . . .
```

Practical 2

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

Hint :

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

Program :

```
int main() {
   // Declare required strings to store parts of the production
   char gram[20];          // Input: the full production RHS, e.g. abcd|abxy
   char part1[20], part2[20]; // Two separate alternatives
   char modifiedGram[20];   // To hold the factored common part + new NT (e.g. aX)
   char newGram[20];        // To hold remaining suffixes (e.g. bcd|bxy)
   char tempGram[20];       // (Unused in this code)

   int i, j = 0, k = 0, l = 0, pos;

   // Input production rule (Only RHS part, assumes LHS is 'A->')
   printf("Enter Production : A->");
   gets(gram); // deprecated, use fgets() in modern code (but okay for now)

   // Step 1: Split into two parts using '|'
   for(i = 0; gram[i] != '|'; i++, j++) {
      part1[j] = gram[i];  // Copy everything before '|' to part1
   }
   part1[j] = '\0';        // Null-terminate part1

   for(j = ++i, i = 0; gram[j] != '\0'; j++, i++) {
      part2[i] = gram[j];  // Copy everything after '|' to part2
   }
   part2[i] = '\0';        // Null-terminate part2

   // Step 2: Find longest common prefix
   for(i = 0; i < strlen(part1) || i < strlen(part2); i++) {
      if(part1[i] == part2[i]) {
         modifiedGram[k++] = part1[i]; // Build the common prefix
         pos = i + 1;                  // Store next index after prefix
      }
   }
```

```
    // Step 3: Get remaining suffixes after the prefix in both parts
    for(i = pos, j = 0; part1[i] != '\0'; i++, j++) {
        newGram[j] = part1[i];  // Remaining part of first production
    }

    newGram[j++] = '|';         // Add separator between productions

    for(i = pos; part2[i] != '\0'; i++, j++) {
        newGram[j] = part2[i];  // Remaining part of second production
    }

    // Step 4: Finalize new grammar strings
    modifiedGram[k] = 'X';       // Add new non-terminal (assumes X is safe)
    modifiedGram[++k] = '\0';    // Null-terminate

    newGram[j] = '\0';           // Null-terminate

    // Step 5: Print result
    printf("\nGrammar Without Left Factoring : \n");
    printf(" A->%s", modifiedGram); // e.g. A->abX
    printf("\n X->%s\n", newGram);  // e.g. X->cd|xy

    return 0;
}
```

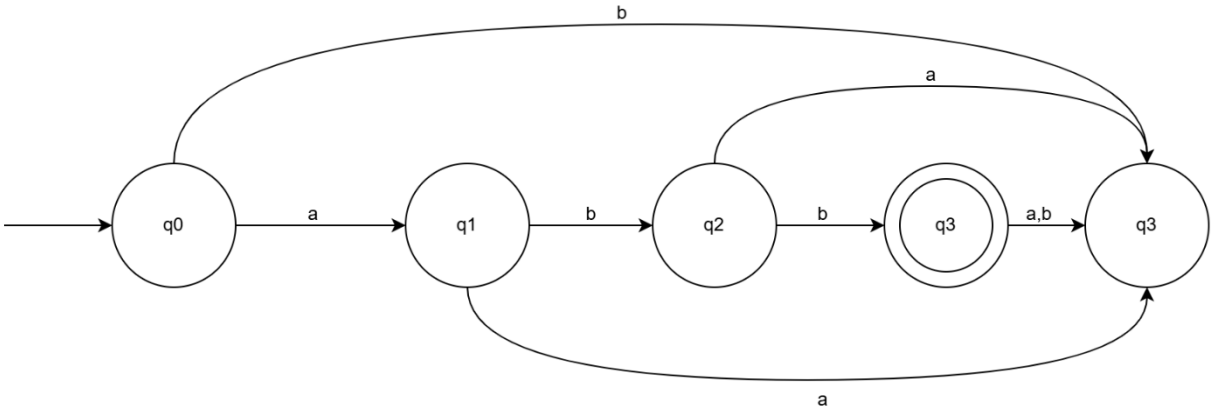
Output:

```
Enter Production : A->abc|abxy

Grammar Without Left Factoring :
 A->abX
 X->c|xy

--------------------------------
Process exited after 12.95 seconds with return value 0
Press any key to continue . . .
```

Practical 3

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

Hint : DFA for accept "abb"



Program :

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

// Automaton using else-if ladder for accepting "abb"
int main() {
    char input[100];
    int state = 0, i = 0;

    printf("Enter a string: ");
    scanf("%s", input);

    while (input[i] != '\0') {
        if (state == 0 && input[i] == 'a') {
            state = 1;
        }
        else if (state == 1 && input[i] == 'b') {
            state = 2;
        }
        else if (state == 2 && input[i] == 'b') {
            state = 3;
        }
        else {
            // Invalid character or unexpected input
            state = -1;
            break;
```

```
        }
        i++;
    }

    // Final state must be 3 for valid "abb"
    if (state == 3 && input[i] == '\0') {
        printf("String ACCEPTED by automaton.\n");
    } else {
        printf("String REJECTED by automaton.\n");
    }

    return 0;
}
```

Output:

```
Enter a string: abb
String ACCEPTED by automaton.

--------------------------------
Process exited after 3.599 seconds with return value 0
Press any key to continue . . .
```

```
Enter a string: xyz
String REJECTED by automaton.

--------------------------------
Process exited after 4.18 seconds with return value 0
Press any key to continue . . .
```

Practical 4

Title: Prepare report for Lex and install Lex on Linux/Windows

LEX in Compiler Design:

Whenever a developer wants to make any software application they write the code in a high-level language. That code is not understood by the machine so it is converted into low-level machine-understandable code by the compiler. Lex is an important part of this compiler and is responsible for the classification of the generated tokens based on their purpose.

Lexical Analysis

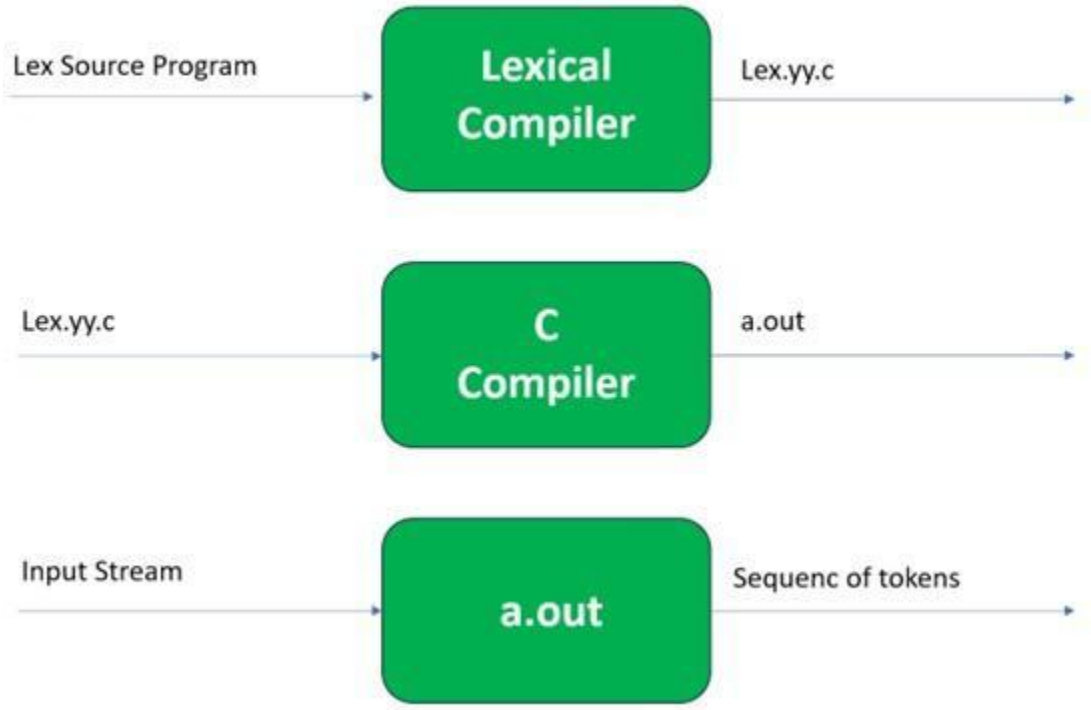
It is the first step of compiler design, it takes the input as a stream of characters and gives the output as tokens also known as tokenization. The tokens can be classified into identifiers, Separators, Keywords, Operators, Constants and Special Characters.

It has three phases:

- Tokenization: It takes the stream of characters and converts it into tokens.
- Error Messages: It gives errors related to lexical analysis such as exceeding length, unmatched string, etc.
- Eliminate Comments: Eliminates all the spaces, blank spaces, new lines, and indentations.

What is Lex in Compiler Design?

- 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.



lex.yy.c: It is a C program.
File.l: It is a Lex source program
a.out: It is a Lexical analyzer

Lex File Format:

A Lex program consists of three parts and is separated by % delimiters: -

Declarations
%%
Translation rules
%%
Auxiliary procedures

Declarations: The declarations include declarations of variables.

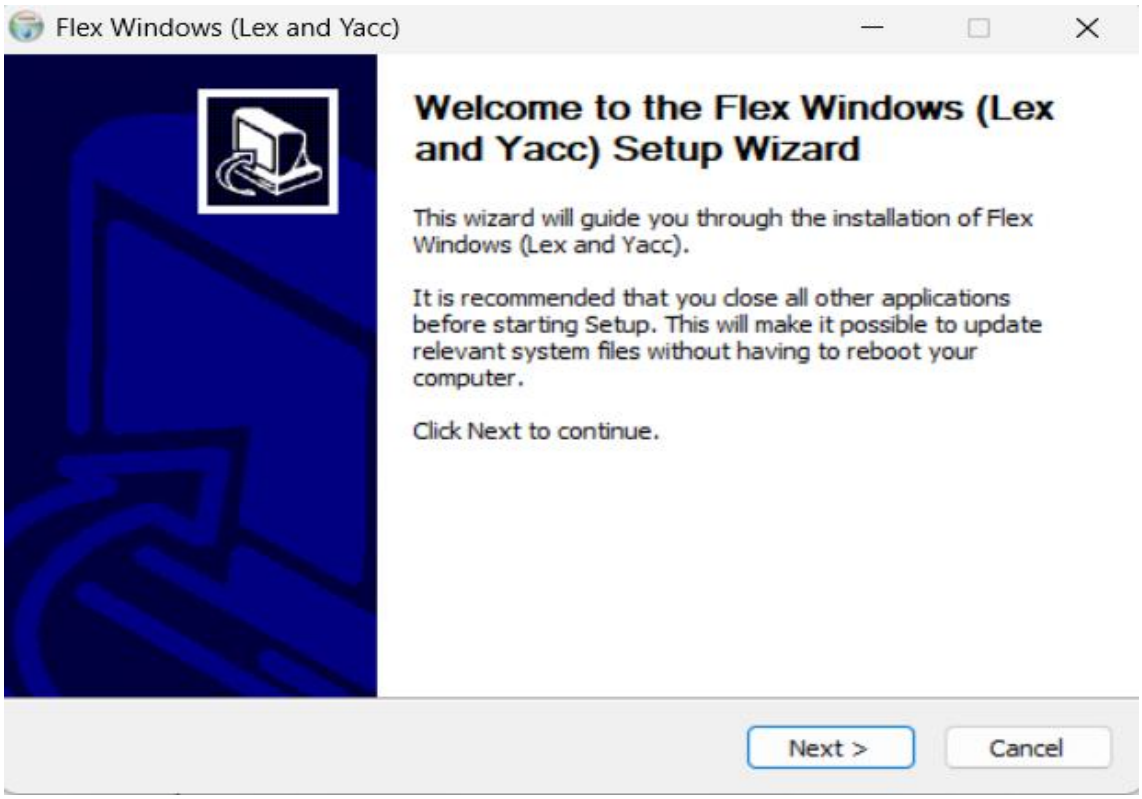
Transition rules: These rules consist of Pattern and Action.

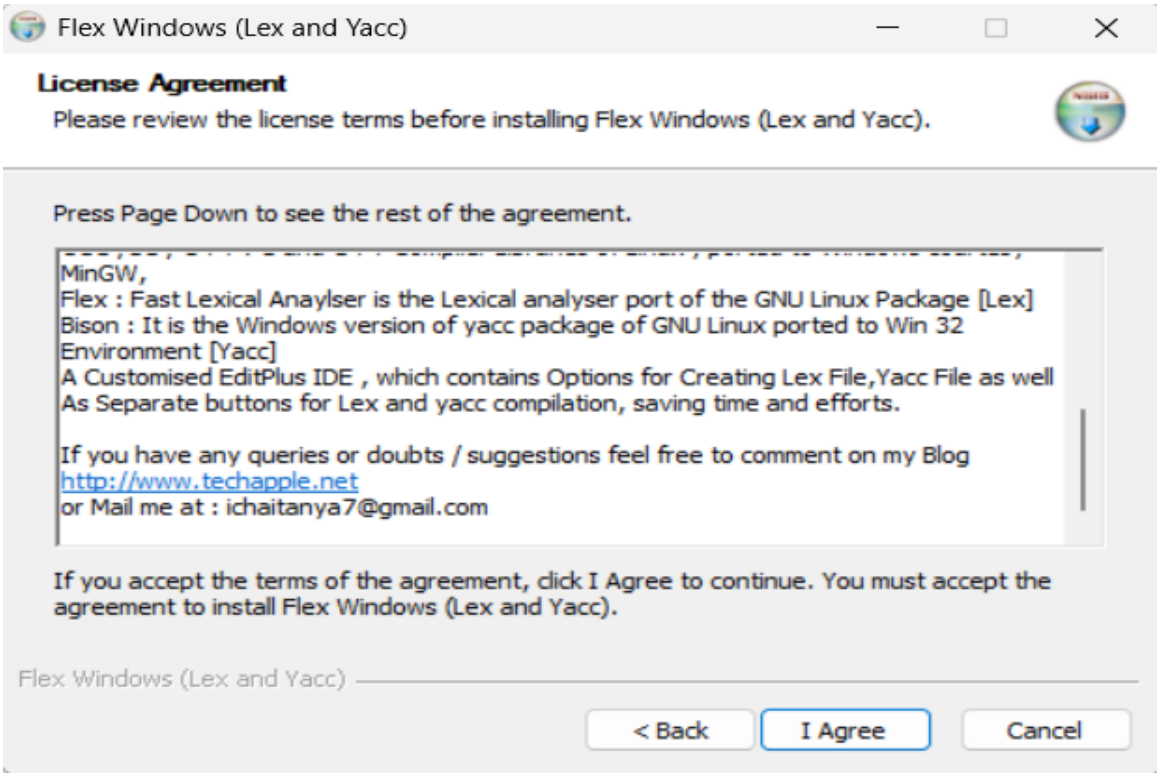
Auxiliary procedures: The Auxiliary section holds auxiliary functions used in the actions.

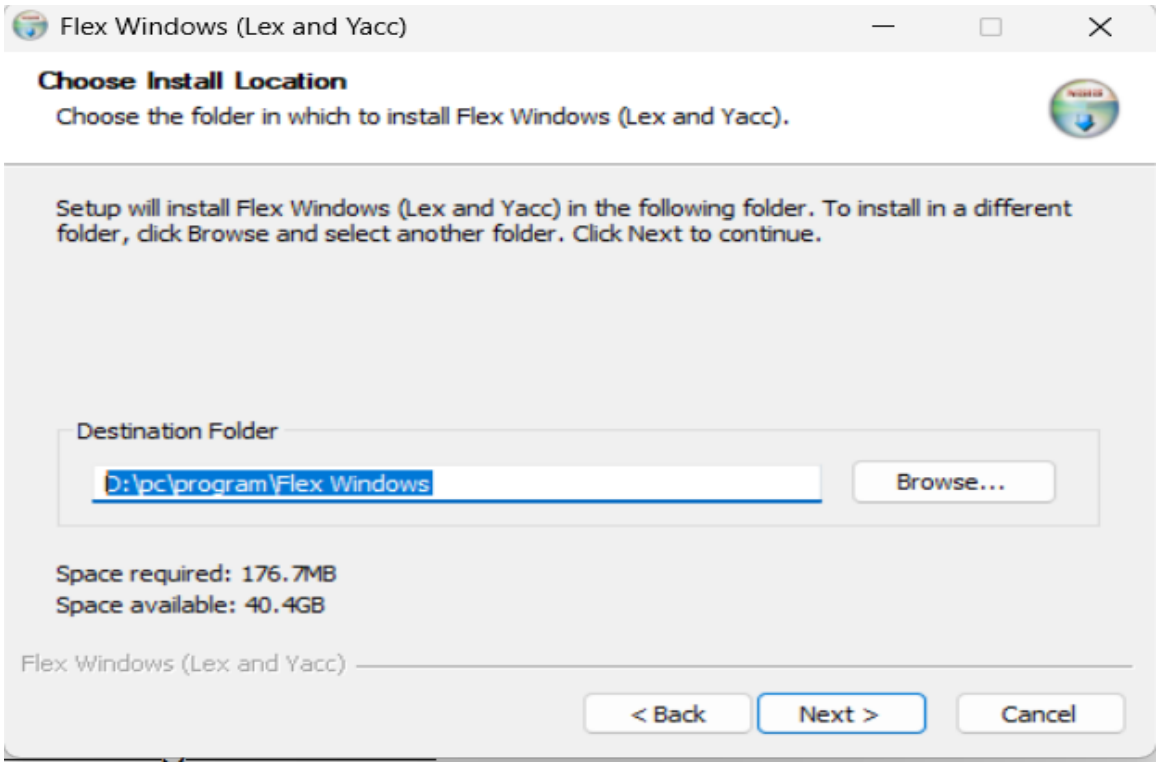
For example:

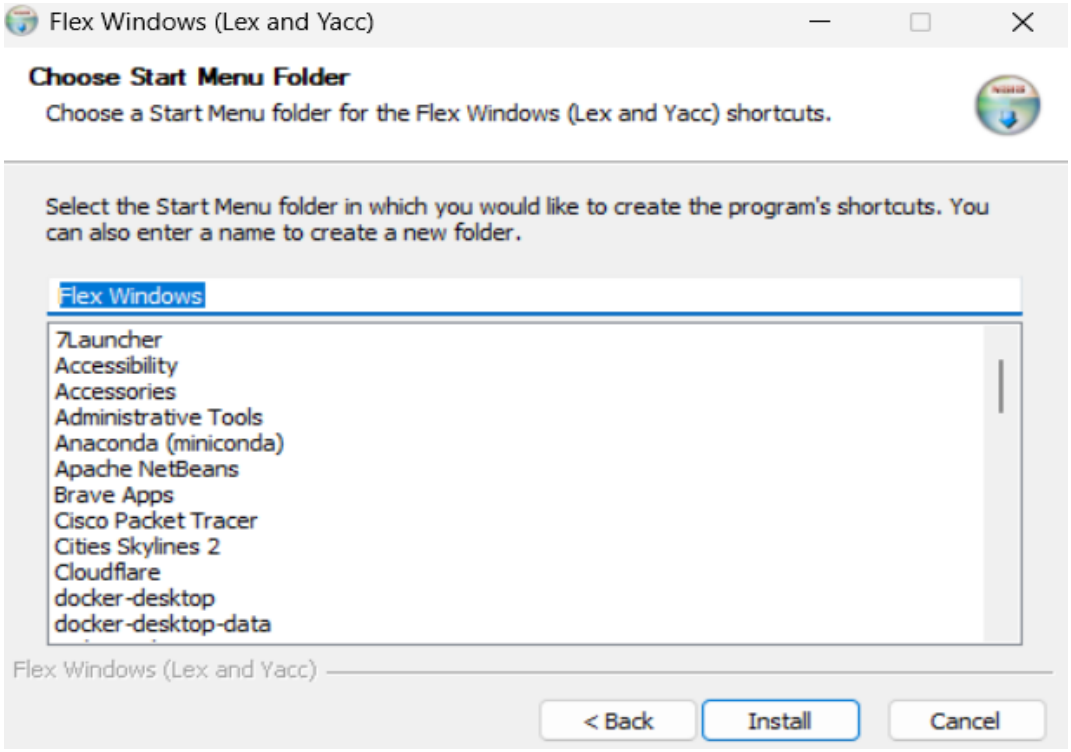
declaration
number[0-9]
%%
translation
if {return (IF);}
%%
auxiliary function
int numberSum()

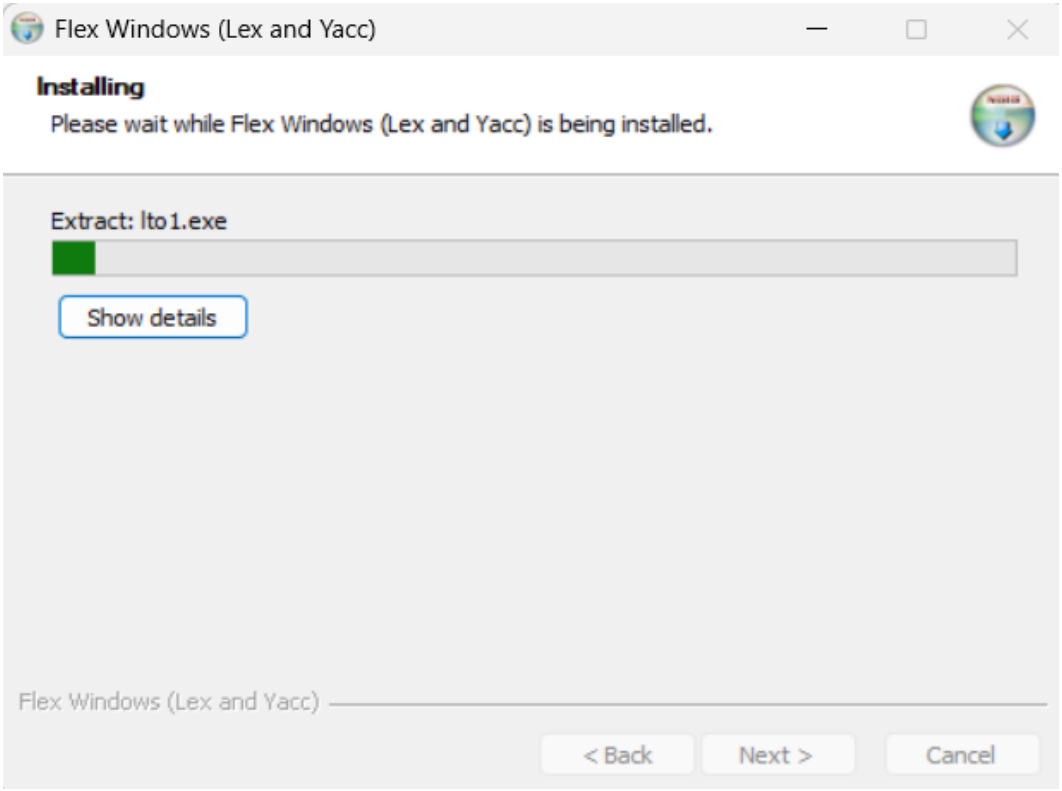
Steps of installing LEX compiler

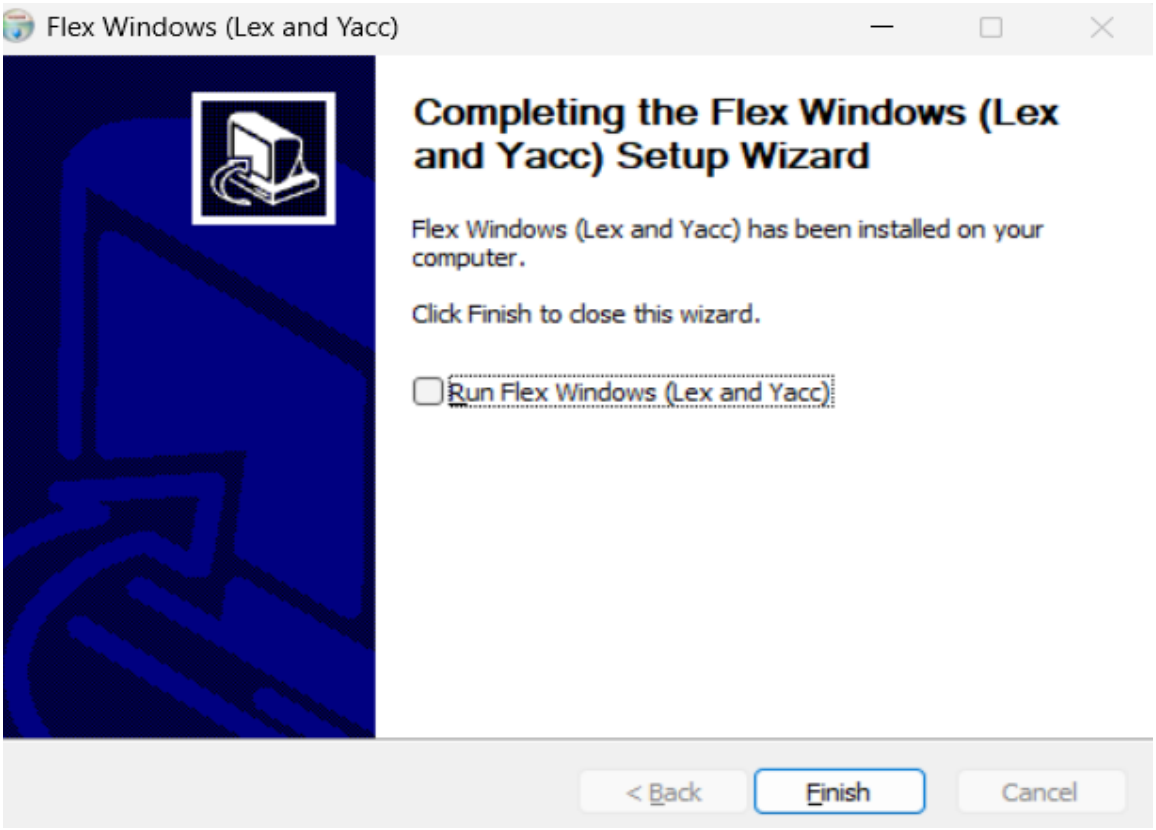



Flex Windows (Lex and Yacc)
License Agreement
Please review the license terms before installing Flex Windows (Lex and Yacc).
Press Page Down to see the rest of the agreement.
MinGW,
Flex : Fast Lexical Anaylser is the Lexical analyser port of the GNU Linux Package [Lex]
Bison : It is the Windows version of yacc package of GNU Linux ported to Win 32 Environment [Yacc]
A Customised EditPlus IDE , which contains Options for Creating Lex File,Yacc File as well As Separate buttons for Lex and yacc compilation, saving time and efforts.
If you have any queries or doubts / suggestions feel free to comment on my Blog
http://www.techapple.net
or Mail me at : ichaitanya7@gmail.com
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Flex Windows (Lex and Yacc).
Flex Windows (Lex and Yacc)
< Back
I Agree
Cancel


Flex Windows (Lex and Yacc)
Choose Install Location
Choose the folder in which to install Flex Windows (Lex and Yacc).
Setup will install Flex Windows (Lex and Yacc) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
D:\pc\program\Flex Windows
Browse...
Space required: 176.7MB
Space available: 40.4GB
Flex Windows (Lex and Yacc)
< Back
Next >
Cancel


Flex Windows (Lex and Yacc)
Choose Start Menu Folder
Choose a Start Menu folder for the Flex Windows (Lex and Yacc) shortcuts.
Select the Start Menu folder in which you would like to create the program's shortcuts. You can also enter a name to create a new folder.
Flex Windows
7Launcher
Accessibility
Accessories
Administrative Tools
Anaconda (miniconda)
Apache NetBeans
Brave Apps
Cisco Packet Tracer
Cities Skylines 2
Cloudflare
docker-desktop
docker-desktop-data
Flex Windows (Lex and Yacc)
< Back
Install
Cancel


Flex Windows (Lex and Yacc)
Installing
Please wait while Flex Windows (Lex and Yacc) is being installed.
Extract: lto1.exe
Show details
Flex Windows (Lex and Yacc)
< Back
Next >
Cancel


Flex Windows (Lex and Yacc)
Completing the Flex Windows (Lex and Yacc) Setup Wizard
Flex Windows (Lex and Yacc) has been installed on your computer.
Click Finish to close this wizard.
Run Flex Windows (Lex and Yacc)
< Back
Finish
Cancel