

# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**CSD 3203 – Compiler Laboratory**

**B.TECH – CSE**

## Name :

RRN :

Semester : VI Semester



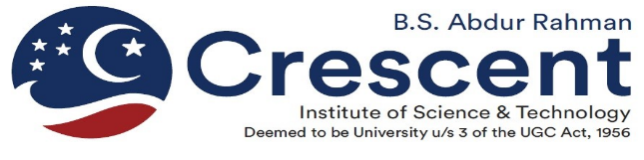
# BONAFIDE CERTIFICATE

Certified that this is the bonafide record of the work done by \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ Register No.of **VI** Semester **B.Tech Computer Science and Engineering** in the **CSD 3203- Compiler Laboratory** during the year **2024.**

Course Faculty Head of theDepartment Date:

Submitted for the Practical Examination held on

Examiner

****

**INDEX**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp.**  **No.** | **Date** | **List of Experiments** | **Aim & Algorithm(10)** | **Implementation**  **(20)** | **Output**  **and Viva**  **(20)** | **Total Marks**  **(50)** | **Sign** |
| **1.** |  | Count number of characters, words and lines in a file |  |  |  |  |  |
| **2.** |  | Count the number of vowels and consonants in a file |  |  |  |  |  |
| **3.** |  | String Operations(any ten without library functions) |  |  |  |  |  |
| **4.** |  | Token Separation using high level language |  |  |  |  |  |
| **5.** |  | Study of LEX Tool |  |  |  |  |  |
| **6.** | LEX Programs | |  |  |  |  |  |
|  |  | Count of positive and negative numbers |  |  |  |  |  |
|  |  | Count of number of words, characters and lines |  |  |  |  |  |
|  |  | Count of vowels and consonants |  |  |  |  |  |
|  |  | Odd or Even |  |  |  |  |  |
|  |  | Mobile Number, Email and date Validation |  |  |  |  |  |
|  |  | String starts with ‘a’ |  |  |  |  |  |
|  |  | Lexical analyzer using LEX Tool |  |  |  |  |  |
| **7.** |  | Study of YACC Tool |  |  |  |  |  |
| **8.** |  | Parsing Techniques using YACC and Lex |  |  |  |  |  |
| **9.** |  | Recursive descent parser |  |  |  |  |  |
| **10.** |  | Scientific Calculator Program using LEX and YACC |  |  |  |  |  |
| **11.** |  | Generate abstract syntax tree and intermediate code |  |  |  |  |  |
| **12.** |  | Three Address Code Generation |  |  |  |  |  |
| **13.** |  | Code Optimizer |  |  |  |  |  |
| **14.** |  | Code Generation |  |  |  |  |  |

**Average:**

**Ex. No.1 Count number of characters, words and lines in a file**

**Date:**

**AIM:** To count the number of characters, words and lines in a file using any high level language.

**ALGORITHM:**

Input: Filename

Output: Character Count, Word Count, Line Count

STEP 1. Set character\_count, word\_count, and line\_count to 0.

STEP 2. Open the file with the given filename in read mode.

STEP 3. Loop through each line in the file:

a. Increment line\_count by 1.

b. Increment character\_count by the length of the current line.

c. Split the line into words using whitespace as a delimiter.

d. Increment word\_count by the number of words obtained.

STEP 4. Close the file.

STEP 5. Output character\_count, word\_count, and line\_count.

**Ex. No.2 Count the number of vowels and consonants in a file**

**Date:**

**AIM:** To count the number of vowels and consonants in a file.

**ALGORITHM:**

Input: Filename

Output: Vowel Count, Consonant Count

STEP 1. Set vowel\_count and consonant\_count to 0.

STEP 2. Define a list or string containing all vowels and consonants in the alphabet.

STEP 3. Open the file with the given filename in read mode.

STEP 4. Loop through each character in the file:

a. Check if the character is an alphabet (ignore case).

b. If it is an alphabet:

i. Check if it is a vowel or a consonant by comparing it with the defined lists of vowels and consonants.

ii. Increment the respective counter (vowel\_count or consonant\_count).

STEP 5. Close the file.

STEP 6. Output vowel\_count and consonant\_count.

**Ex. No.3 String Operations (any ten without library functions)**

**Date:**

**AIM:** To implement string operations using any high level language.

**ALGORITHM:**

STEP 1. \*\*Input String\*\*: Take the input string from the user or from a file.

STEP 2. \*\*String Length\*\*:

- Initialize a variable `length` to 0.

- Iterate through each character in the string.

- Increment `length` by 1 for each character.

- Output the value of `length`.

STEP 3. \*\*Concatenation\*\*:

- Take another string as input (if needed).

- Concatenate the input string with the second string.

- Output the concatenated string.

STEP 4. \*\*Substring Extraction\*\*:

- Take input for starting index `start` and ending index `end`.

- Extract the substring from `start` to `end`.

- Output the substring.

STEP 5. \*\*String Reversal\*\*:

- Initialize an empty string `reversed\_string`.

- Iterate through the input string in reverse order.

- Append each character to `reversed\_string`.

- Output `reversed\_string`.

STEP 6. \*\*Search Substring\*\*:

- Take a substring as input.

- Iterate through the input string.

- Check if the substring is present at each index using string slicing or other methods.

- If found, output the index; otherwise, output a message indicating that the substring is not found.

STEP 7. \*\*Replace Substring\*\*:

- Take two substrings as input: the substring to be replaced (`old\_substring`) and the new substring (`new\_substring`).

- Use string manipulation methods to replace `old\_substring` with `new\_substring`.

- Output the modified string.

STEP 8. \*\*Count Occurrences\*\*:

- Take a substring as input.

- Initialize a counter variable to 0.

- Iterate through the input string.

- Increment the counter each time the substring is found.

- Output the value of the counter.

STEP 9. \*\*Convert Case\*\*:

- Take input for choosing the case conversion (e.g., lowercase to uppercase, uppercase to lowercase).

- Use built-in functions or methods to convert the case of the string.

- Output the converted string.

STEP 10. \*\*Exit\*\*:

- Provide an option to exit the program.

**Ex. No.4 Token Separation using high level language**

**Date:**

**AIM:** To implement token Separation using high level language.

**ALGORITHM:**

Certainly! Below is a generic algorithm to implement token separation using any high-level language:

STEP 1. \*\*Input String\*\*: Take the input string from the user or from a file.

STEP 2. \*\*Define Delimiters\*\*: Define a set of characters that will be used as delimiters to separate tokens. Common delimiters include whitespace, commas, periods, semicolons, etc.

STEP 3. \*\*Tokenization Process\*\*:

- Initialize an empty list or array to store the tokens.

- Initialize variables `start\_index` and `end\_index` to keep track of the positions of tokens within the input string.

- Iterate through each character in the input string.

- If the character is a delimiter:

- If `start\_index` is not equal to `end\_index`, extract the token from the input string starting from `start\_index` to `end\_index`, and append it to the list of tokens.

- Update `start\_index` to the position of the next character after the delimiter.

- If the character is not a delimiter:

- If `start\_index` is equal to `end\_index`, update `start\_index` to the current position of the character.

- Update `end\_index` to the current position of the character.

- After the loop, if `start\_index` is not equal to `end\_index`, extract the last token from the input string and append it to the list of tokens.

- Output the list of tokens.

STEP 4. \*\*Exit\*\*:

- Provide an option to exit the program.

**Ex. No.5 Study of LEX Tool**

**Date:**

**AIM:** To study about lex tool and implement simple program using lex code.

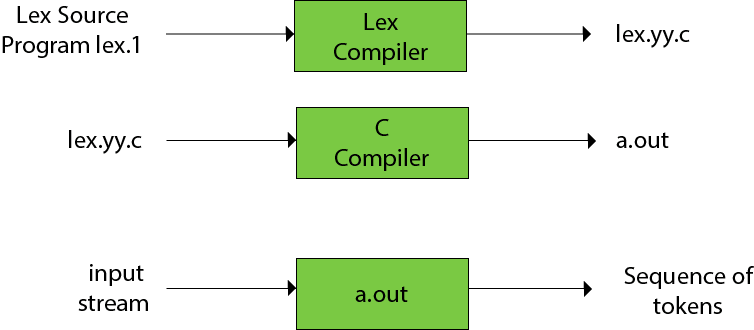
**DESCRIPTION & PROCEDURE:**

**Introduction**

* Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
* The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
* It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

* Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
* Finally C compiler runs the lex.yy.c program and produces an object program a.out.
* a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



**Lex file format**

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }

**Definitions** include declarations of constant, variable and regular definitions.

**Rules** define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

**To run a lex program**

* Save the program with the extension .l or .lex
* To compile the program
* In telnet screen, > lex filename.l
* Compile:

>cc lex.yy.c -ll

* To execute the program

>./a.out

* Provide the required input to the program

Simple Lex Program: simple.l

%%

Welcome {printf(“\n welcome to Lexical Program”);}

%%

To compile and execute:

>lex simple.l

>cc lex.yy.c –ll

>./a.out

welcome

welcome to Lexical Program

**Ex. No.6a LEX Programs-** Count of positive and negative numbers

**Date:**

**AIM:** To write LEX code to count positive and negative numbers.

**ALGORITHM:**

STEP 1. Define rules for positive and negative numbers:

- A positive number can start with a digit other than zero, followed by zero or more digits.

- A negative number starts with a minus sign '-' followed by a digit other than zero, followed by zero or more digits.

STEP 2. Write Lex code to recognize these patterns:

- Use regular expressions to define patterns for positive and negative numbers.

- Write corresponding actions to increment counters for positive and negative numbers.

STEP 3. Maintain counters to count the occurrences of positive and negative numbers:

- Initialize two counters, one for positive numbers and one for negative numbers.

STEP 4. Output the counts:

- After scanning the input, output the counts of positive and negative numbers.

**Ex. No.6b LEX Programs-** Count of number of words, characters and lines

**Date:**

**AIM:** To write LEX code to count of number of words, characters and lines.

**ALGORITHM:**

STEP 1. Define rules for recognizing words, characters, and lines:

- A word is defined as a sequence of characters separated by whitespace (spaces, tabs, or newlines).

- A character can be any printable character including whitespace.

- A line is defined as a sequence of characters terminated by a newline character.

STEP 2. Write Lex code to tokenize the input text:

- Use regular expressions to define patterns for words, characters, and lines.

- Write corresponding actions to increment counters for words, characters, and lines.

STEP 3. Maintain counters to count the occurrences of words, characters, and lines:

- Initialize three counters: one for words, one for characters, and one for lines.

STEP 4. Output the counts:

- After scanning the input, output the counts of words, characters, and lines.

**Ex. No.6c LEX Programs-** Count of vowels and consonants

**Date:**

**AIM:** To write LEX code to count of vowels and consonants.

**ALGORITHM:**

STEP 1. Define rules for recognizing vowels and consonants:

- Vowels are typically the letters 'a', 'e', 'i', 'o', 'u' (both lowercase and uppercase).

- Consonants are all other letters excluding vowels and non-alphabetic characters.

STEP 2. Write Lex code to tokenize the input text:

- Use regular expressions to define patterns for vowels and consonants.

- Write corresponding actions to increment counters for vowels and consonants.

STEP 3. Maintain counters to count the occurrences of vowels and consonants:

- Initialize two counters, one for vowels and one for consonants.

STEP 4. Output the counts:

- After scanning the input, output the counts of vowels and consonants.

**Ex. No.6d LEX Programs-** Odd or Even

**Date:**

**AIM:** To write LEX code to identify if number is odd or even.

**ALGORITHM:**

STEP1. Define rules for recognizing numbers:

- A number can start with an optional negative sign '-' followed by one or more digits.

STEP 2. Write Lex code to tokenize and identify the numbers:

- Use regular expressions to define patterns for numbers.

- Write corresponding actions to extract and identify the numbers.

STEP 3. Check if the number is odd or even:

- After identifying the number, convert it to an integer.

- Check if the number is divisible by 2:

- If divisible, it's an even number.

- If not divisible, it's an odd number.

STEP 4. Output the result:

- Print whether the number is odd or even.

**Ex. No.6e LEX Programs-** Mobile Number, Email and date Validation

**Date:**

**AIM:** To write LEX code to validate mobile number, Email and date.

**ALGORITHM:**

STEP 1. Define rules for recognizing valid mobile numbers, email addresses, and dates:

- Valid mobile numbers typically consist of digits and may include country codes, prefixes, and separators like hyphens or spaces.

- Valid email addresses consist of a local part followed by an "@" symbol and a domain part.

- Valid dates can have various formats such as "YYYY-MM-DD", "MM/DD/YYYY", "DD-MM-YYYY", etc.

STEP 2. Write Lex code to tokenize and identify these patterns:

- Use regular expressions to define patterns for mobile numbers, email addresses, and dates.

- Write corresponding actions to extract and identify these patterns.

STEP 3. Implement validation checks for each pattern:

- For mobile numbers, check if the extracted pattern matches the expected format and includes valid digits.

- For email addresses, verify if the local part and domain part are valid.

- For dates, validate the format and check if the extracted values for year, month, and day are within valid ranges.

STEP 4. Output whether the input matches the valid pattern or not:

- Print whether the input matches the valid format for mobile number, email, or date.

- If not valid, print an error message indicating the reason for invalidation.

**Ex. No.6f LEX Programs-** **String starts with ‘a’**

**Date:**

**AIM:** To write LEX code to identify the string starts with ‘a’.

**ALGORITHM:**

STEP 1. Define rules for recognizing strings that start with 'a':

- A string starts with 'a' if the first character is 'a'.

STEP 2. Write Lex code to tokenize and identify these patterns:

- Use regular expressions to define patterns for strings.

- Write corresponding actions to extract and identify these patterns.

STEP 3. Output the strings that match the specified pattern:

- Print or output the strings that start with 'a'.

**Ex. No.6g Lexical analyzer using LEX Tool**

**Date:**

**AIM:** To implement Lexical analyzer using LEX Tool.

**ALGORITHM:**

STEP 1. Define tokens and their corresponding regular expressions:

- Identify the different types of tokens in the input text (e.g., keywords, identifiers, operators, literals, etc.).

- Define regular expressions to recognize these tokens.

STEP 2. Write Lex code to recognize these tokens:

- Use Lex syntax to define rules for recognizing tokens based on the regular expressions.

- Write corresponding actions to perform any necessary processing for each token.

STEP 3. Compile the Lex code to generate the lexical analyzer program:

- Use the Lex tool to compile the Lex code and generate C code.

- Compile the generated C code using a C compiler to create the lexical analyzer executable.

STEP 4. Use the generated program to analyze input text:

- Provide input text to the lexical analyzer executable.

- The lexical analyzer will tokenize the input text, identify tokens, and perform any specified actions.

**Ex. No.7 Study of YACC Tool**

**Date:**

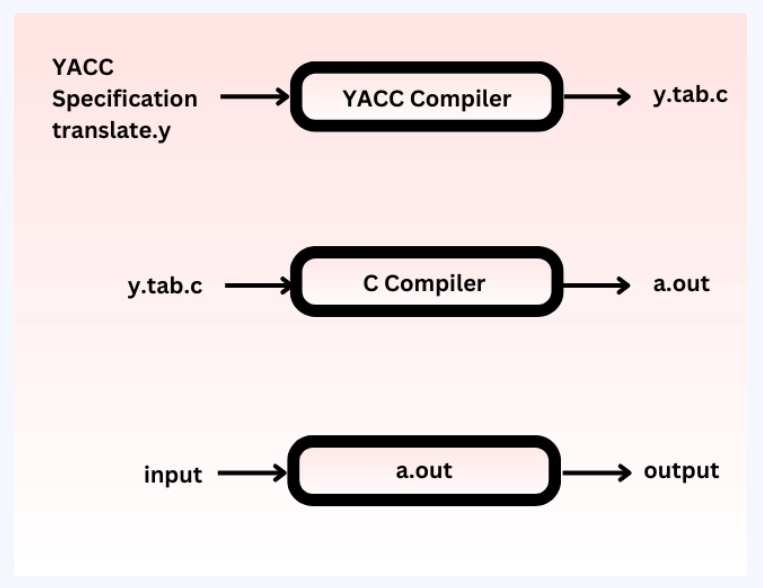
**AIM:** To study about YAAC tool and implement simple program.

**DESCRIPTION & PROCEDURE:**

YACC(Yet Another Compiler Compiler)," serves as a powerful grammar parser and generator. In essence, it functions as a tool that takes in a grammar specification and transforms it into executable code capable of meticulously structuring input tokens into a coherent syntactic tree, aligning seamlessly with the prescribed grammar rules.

Stephen C. Johnson developed YACC in compiler design in the early 1970s. Initially, the YACC was written in the B programming language and was soon rewritten in C. It was originally designed for being complemented by Lex.

In addition to that, YACC was also rewritten in OCaml, Ratfor, ML, ADA, Pascal, Java < Python, Ruby and Go.

****

The input of YACC in compiler design is the rule or grammar, and the output is a C program.

## Parts of a YACC Program in Compiler Design

The parts of YACC program are divided into three sections:

/\* definitions \*/

....

%%

/\* rules \*/

....

%%

/\* auxiliary routines \*/

....

**Definitions:** these include the header files and any token information used in the syntax. These are located at the top of the input file. Here, the tokens are defined using a modulus sign. In the YACC, numbers are automatically assigned for tokens.

Let us see some examples:

%token ID

{% #include <stdio.h> %}

**Rules:**The rules are defined between *%%* and *%%.* These rules define the actions for when the token is scanned and are executed when a token matches the grammar.

**Auxiliary Routines:**Auxiliary routines contain the function required in the rules section. This Auxiliary section includes the *main()* function, where the *yyparse()* function is always called.

This *yyparse()* function plays the role of reading the token, performing actions and then returning to the main() after the execution or in the case of an error.

*0* is returned after successful parsing and *1* is returned after an unsuccessful parsing.

The YACC is responsible for converting these sections into subroutines which will examine the inputs. This process is made to work by a call to a low-level scanner and is named Parsing

Let us now study the working of YACC in compiler design.

## **Workings of YACC**

YACC in compiler design is set to work in C programming language along with its parser generator.

* An input with a .y extension is given.
* The file is invoked, and 2 files, y.tab.h and y.tab.c, are created. These files contain long codes implementing the LARl (1) Parser for grammar.
* This file then provides yyparse.y, which tries to parse a valid sentence successfully.

For the output files,

* If called with the –d option in the command line, YACC produces y.tab.h with all its specific definitions.
* If called with the –v option, YACC produces y.output having a textual description of the LALR(1) parsing table.

### Compile and Execute

* Write lex program in a file file. l and yacc in a file file. y.
* Open Terminal and Navigate to the Directory where you have saved the files.

>lex file. l.

>type yacc file. y.

>type cc lex. yy. c y. tab. h -ll.

>type ./a. out.

**SIMPLE PROGRAM TO VALIDATE STRING**

**ALGORITHM:**

1. Define the grammar for the valid structure of the string using BNF (Backus-Naur Form) notation.
2. Write Yacc code to implement the grammar rules and associated actions to validate the string.
3. Use Lex to provide tokenization for Yacc.
4. Compile the Yacc code to generate the parser.
5. Use the generated parser to validate input strings

**SOURCE CODE:**

simple.l(lex file)

%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

%%

0 {yylval=0; return z;}

1 {yylval=1; return o;}

[ \t] {;}

\n {return 0;}

. return yytext[0];

%%

int yywrap()

{

return 1;

}

test.y(yacc file)

%{

#include<stdio.h>

#include<stdlib.h>

void yyerror();

%}

%token z o

%%

Stmt : S | T ;

S : S C | C;

T : T D | D;

C : z z;

D : o o;

%%

void main()

{

printf("Enter the string:");

yyparse();

printf("Accepted\n");

exit(0);

}

void yyerror(char \*S)

{

printf("Not accepted\n");

exit(0);

}

OUTPUT:

>lex simple.l

>yacc test. y.

>cc lex. yy. c y. tab. h -ll.

>./a. out.

Enter the string:0011

Not accepted

> ./a.out

Enter the string:1111

Accepted

> ./a.out

Enter the string:000

Not accepted

**Ex. No. 8 Parsing Techniques using YACC and Lex**

**Date:**

**AIM:** To implement parsing techniques using YACC and Lex code.

**ALGORITHM:**

STEP 1. Define the grammar for the language you want to parse using BNF notation:

- Identify the structure and rules that define the language.

- Define production rules for terminals (tokens) and non-terminals (components of the language).

STEP 2. Write Lex code to tokenize input text into tokens based on the grammar rules:

- Use regular expressions to define patterns for tokens.

- Associate each token with a token type.

STEP 3. Write Yacc code to define parsing rules according to the grammar and associate actions with each rule:

- Use Yacc syntax to define grammar rules based on the BNF notation.

- Write corresponding actions to perform parsing and processing of input based on the grammar rules.

STEP 4. Compile the Lex and Yacc code to generate the parser:

- Use the Lex and Yacc tools to compile the Lex and Yacc code, respectively.

- Generate C code from the Lex and Yacc files.

- Compile the generated C code using a C compiler to create the parser executable.

STEP 5. Use the generated parser to parse input text and perform desired actions:

- Provide input text to the parser executable.

- The parser will tokenize the input text using Lex, parse it according to the grammar rules using Yacc, and perform actions specified in the Yacc code.