Faculty of Technology and Engineering

Computer Engineering / Computer Science and Engineering

Date : 10 / 12 / 2025

Practical List

Academic Year	:	2025-2026	Semester	:	6
Course code	:	CSE313	Course name	:	Design of Language Processor

Sr. No.	Aim	Hr	CO
1.	**Practical Definition** String Validation Against Regular Expression **Objective** To implement a program that validates a user-input string against the regular expression a*bb. The program should determine whether the input string is valid or invalid based on the defined pattern. **Language Constraint** The program must be implemented in C language. **Input Requirements** • Accept a character string from the user. • Ensure the input is terminated with a newline character. **Expected output** • If the input string matches the pattern a*bb, the program should output: "Valid String". • If the input string does not match the pattern, the program should output: "Invalid String". **Sample input output** Input: aaabb → Valid string; Abab → Invalid string **Testcases** ^ bbbb aaa baaabb aaabbb baaabb aaaab abbabb abb aaaaabb	2	1

2.	**Practical Definition** String Validation Using Finite Automata **Objective** To implement a program that validates a given string against rules defined in terms of finite automata. **Language Constraint** The program can be implemented in any programming language **Input requirement** • Accept rules in the form of finite automata (e.g., states, transitions, start state, accept states) as input. • Accept a string to be validated against the provided finite automata rules. **Expected output** • If the string adheres to the rules of the finite automata, the program should output: "Valid String". • If the string does not adhere to the rules, the program should output: "Invalid String". **Sample input output** Input: Number of input symbols : 2 Input symbols : a b Enter number of states : 4 Initial state : 1 Number of accepting states : 1 Accepting states : 2 Transition table : 1 to a -> 2 1 to b -> 3 2 to a -> 1 2 to b -> 4 3 to a -> 4 3 to b -> 1 4 to a -> 3 4 to b -> 2 Input string : abbabab Output: Valid string **Testcases** • String over 0 and 1 where every 0 immediately followed by 11 • string over a b c, starts and end with same letter. • String over lower-case alphabet and digits, starts with alphabet only.	2	1

3.	**Practical Definition** Implementation of a Lexical Analyzer for C Language Compiler **Objective** To design and implement a lexical analyser, the first phase of a compiler, for the C programming language. The lexical analyser should perform the following tasks: (1) tokenizing the input string (2) removing comments (3) removing white spaces (4) entering identifiers into the symbol table (5) generating lexical errors. **Language Constraint** The program can be implemented in any programming language **Input requirement** • Accept a C source code file. • The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces. **Expected output** • Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator. • Symbol table with all identified identifiers stored. • Detection and reporting of lexical errors • Modified source code **Sample input output**	4	1

Input	Output	

int main() {
int a = 5 , 7H;
// assign value
char b = 'x';
/* return
value */
return a + b;
}
	TOKENS
	Keyword: int
	Identifier: main
	Punctuation: (
	Punctuation:)
	Punctuation: {
	Keyword: int
	Identifier: a
	Operator: =
	Constant: 5
	Punctuation : ,
	Punctuation: ;
	Keyword: char
	Identifier: b
	Operator: =
	String: 'x'
	Punctuation: ;
	Keyword: return
	Identifier: a
	Operator: +
	Identifier: b
	Punctuation: ;
	Punctuation: }
		LEXICAL ERRORS
		7H invalid lexeme
		
		SYMBOL TABLE ENTRIES
		1) a
		2) b

Testcases

/* salary calculation*/ void main( )	// user defined data type struct student

		{
		long int bs , da , hra , gs;
		//take basic salary as input
		scanf("%ld",&bs);
		//calculate allowances
		da=bs*.40;
		hra=bs*.20;
		gs=bs+da+hra;
		// display salary slip
		printf("\n\nbs : %ld",bs);
		printf("\nda : %ld",da);
		printf("\nhra : %ld",hra);
		printf("\ngs : %ld",gs);
		}
			{
			int id;
			float cgpa;
			}
			void main( )
			{
			student s;
			s.id = 10;
			s.cgpa = 8.7;
			}		

		//function prototype
		void add (int , int);
		void main( )
		{
		int a , b;
		a = 10;
		b = 20;
		// function call
		add (a , b);
		}
		void add (int x , int y)
		{
		return x + y;
		}			

4. **Practical Definition** — 4 — 1

String validation using Lax tool

Objective - 1

Write a program to identify and extract all numbers from input string and display them one by one in new line.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

- Accept a character string, mix of text and numbers, from the user.
- Ensure the input is terminated with a newline character.

Expected output

The program should print out each number found in the input, each on a new line.

Sample input output

Input	Output
a1b22c3	1 22 3

Testcases

power operation -> 12 ** 3 = 1728
You multiply 804569 with 1 then will be :

Objective - 2

Write a program to replace the word "charusat" with “university” in the input text.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

- Accept a character string from the user where the word "charusat" may appear multiple times.
- Ensure the input is terminated with a newline character.

Expected output

The program should print the input text with all occurrences of "charusat" replaced by “university”.

Sample input output

Input	Output
This is charusat.	This is university.

Testcases

Charusat is in Anand district.	I am doing my BTech from CHARSAT.
Charusat , What is charusat?	Every where it is charusat , charusat and only charusat.

Objective – 3
Write a program to count number of characters, word and lines from the input file.

Language Constraint
Lex (Lexical analyser generator)

Input requirement
Read contain from a text file containing multiple word and lines.

Expected output
The program should print total number of characters (including spaces), words (separated by white spaces), lines (end with new line symbol).

Sample input output

Input	Output
The 45 is odd number.	Characters : 22 Words : 5 Line : 1

Testcases

I want to calculate a number. The number of characters, words and lines. All know that \n is ending character of line.
45 + 89 =40

Objective – 4
Write a program which validate the password as per given rules.
- length can be 9 to 15 characters
- includes lower case letter, upper case letter, digit, symbols (*, ; # $ @)
- minimum count for each category must be one

Language Constraint
Lex (Lexical analyser generator)

Input requirement
- Accept a character string from the user which is mix of letters, numbers and symbols.
- Ensure the input is terminated with a newline character.

Expected output
- If the password meets the given rules, the program should print "Valid password".
- If the password does not meet the rules, the program should print "Invalid password".

Sample input output

Input	Output
a@1T	Invalid password

Testcase

aB1@	aaBB11,#cdefg2345	CHARUSAT	
Charusat	CHArusat123	Cspit-2024	
Charusat@2024	Charu$at@20#24	charu*sAT;22	

5.

Practical Definition

Implementation of a Lexical Analyzer for C Language Compiler

Objective

To design and implement a lexical analyser to perform 1st, 2nd, 3rd, and 5th task as per the list given in practical 2.

Language Constraint

Lex (Lexical analyser generator)

Input requirement

- Accept a C source code file.
- The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output

- Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.
- Detection and reporting of lexical errors

Sample input output

Input	Output

int main() {
int a = 5 , 7H;
// assign value
char b = 'x';
/* return
value */
return a + b;
}
	TOKENS
	Keyword: int
	Identifier: main
	Punctuation: (
	Punctuation:)
	Punctuation: {
	Keyword: int
	Identifier: a
	Operator: =
	Constant: 5
	Punctuation : ,
	Punctuation: ;
	Keyword: char
	Identifier: b
	Operator: =
	String: 'x'
	Punctuation: ;
	Keyword: return
	Identifier: a
	Operator: +
	Identifier: b
	Punctuation: ;
	Punctuation: }

Testcases

/* salary calculation*/
void main( )
{
long int bs , da , hra , gs;
//take basic salary as input
scanf("%ld",&bs);
//calculate allowances
	// user defined data type
	struct student
	{
	int id;
	float cgpa;
	}
	void main( )

2 | 1

		da=bs*.40;
		hra=bs*.20;
		gs=bs+da+hra;
		// display salary slip
		printf("\n\nbs : %ld",bs);
		printf("\nda : %ld",da);
		printf("\nhra : %ld",hra);
		printf("\ngs : %ld",gs);
		}
			{
			student s;
			s.id = 10;
			s.cgpa = 8.7;
			}		

		//function prototype
		void add (int , int);
		void main( )
		{
		int a , b;
		a = 10;
		b = 20;
		// function call
		add (a , b);
		}
		void add (int x , int y)
		{
		return x + y;
		}			

6.	**Practical definition** String validation using Recursive Descent Parsing (RDP) **Objective** Implement a Recursive Descent Parser (RDP) to validate an input string against the given grammar. S → (L) | a L → S L' L' → , S L' | ϵ **Language constraint** The program can be implemented in any programming language **Input Requirement** A string that needs to be validated against the grammar **Expected output** • If the input string is valid according to the grammar, the program should print "Valid string". • If the input string is invalid according to the grammar, the program should print "Invalid string". **Sample input output** Input: (a) — Output: Valid string **Testcases** a, (a), (a,a), (a,(a,a),a), (a,a),(a,a) a), (a, a,a, a,, (a,a),a	2	2

7.	**Program definition** Computing First and Follow Sets for a Context-Free Grammar (CFG) **Objective** Develop a program computes the First and Follow sets for all non-terminal symbols in for the below given grammar. S → A B C | D A → a | ε B → b | ε C → (S) | c D → A C **Language Constraint** The program can be implemented in any programming language **Input requirement** No input **Expected output** First(S) = {a, b, (, c} First(A) = {a, ε} First(B) = {b, ε} First(C) = {(, c} First(D) = {a, (} Follow(S) = {), $} Follow(A) = {b, (,), $} Follow(B) = {c,), $} Follow(C) = {), $} Follow(D) = {), $}	2	2

8.	**Program definition** Predictive Parsing Table Construction and LL(1) Grammar Validation **Objective** Develop a program to construct a predictive parsing table for the given grammar. The program should analyse the generated parsing table to determine whether the grammar is LL(1) or not. If the grammar is LL(1), the program should also validate an input string against the given grammar. **Language Constraint** The program can be implemented in any programming language **Input requirement** • First() and Follow() generated by practical 7 • A string that needs to be validated against the grammar **Expected output** • A predictive parsing table generated for the given grammar. • A message indicating whether the grammar is LL(1) or not. • If the input string is valid according to the grammar, the program should print "Valid string". • If the input string is invalid according to the grammar, the program should print "Invalid string". **Testcases** abc | ac | (abc) | c | (ac) a | ( ) | (ab) | abcabc | b	3	2

9.

Program definition
String parsing using YACC

Objective
Develop a YACC program to validate input strings based on the given grammar. The program should parse the string using the grammar rules and determine whether the string is valid or invalid.
$S \rightarrow i\ E\ t\ S\ S' \mid a$
$S' \rightarrow e\ S \mid \varepsilon$
$E \rightarrow b$

Language Constraint
YACC (Syntex Analyser generator)

Input requirement
An input string to validate based on the provided grammar.

Expected output
- If the input string is valid according to the grammar, the program should print "Valid string".
- If the input string is invalid according to the grammar, the program should print "Invalid string".

Sample input output

Input	Output
i b t	Invalid string

Testcases

ibtai	ibtaea	a	ibtibta	ibtaeibta
ibti	ibtaa	iea	ibtb	ibtibt

3

2

10.	**Program definition** Evaluating Arithmetic Expression with Bottom-Up Approach Using SDD **Objective** Develop a program to evaluate arithmetic expressions containing operators using a bottom-up parsing approach and below given Syntax-Directed Definitions (SDD) for semantic evaluation. The program will compute the result of the expression by building a parse tree using and will incorporate semantic rules to evaluate sub-expressions during parsing.	2	4

L → E n	Print (E.val)
E → E + T	E.Val = E.val + T.val
E → E - T	E.Val = E.val - T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → T / F	T.val = T.val / F.val
T → F	T.val = F.val
F → G ^ F	F.val = G.val ^ F.val
F → G	F.val = G.val
G → (E)	G.val = E.val
G → digit	G.val = digit.lexval

Language Constraint
An input string

Input requirement
An arithmetic expression in the form of a string that can contain

- Operands: Integers (e.g., 3, 5, 10) or decimals (e.g., 2.5, 0.75)
- Operators: +, -, *, /, ^ for addition, subtraction, multiplication, division, and exponentiation
- Parentheses: (and) for grouping sub-expressions

Expected output

- The evaluated result of the arithmetic expression.
- If the input expression is invalid, the program should display “Invalid expression”.

Sample input output

Input	Output
(3 + 5) * 2 ^ 3	64

Testcases

(3 + 5) * 2	3 + 5 * 2	3 + 5 * 2 ^ 2	3 + (5 * 2)	3 + 5 ^ 2 * 2
3 * (5 + 2)	(3 + 5) ^ 2	3 ^ 2 ^ 3	3 ^ 2 + 5 * 2	3 + ^ 5
(3 + 5 * 2	(3 + 5 * 2 ^ 2 - 8) / 4 ^ 2 + 6			

11.	**Program definition** Generate Three-Address Code (TAC) for an Arithmetic Expression **Objective** Develop a program that parses an input arithmetic expression and generates its equivalent Three-Address Code (TAC) using temporary variables for intermediate results. **Language Constraint** The program can be implemented in any programming language. **Input requirement** The program must accept an arithmetic expression in string format, which may include: • **Operands:** integers (e.g., 4, 15), decimals (e.g., 3.5), or variables (e.g., x, y, total) • **Operators:** +, –, *, /, ^ • **Parentheses:** for grouping sub-expressions **Expected output** A sequence of Three-Address Code (TAC) instructions of the form: t1 = operand1 op operand2 t2 = t1 op operand **Sample input output** **Input** **a + b * c – d** **Out Put** **t1 = b * c** **t2 = a + t1** **t3 = t2 - d** **Test cases** **Test case – 1** **(5 + 3) * 2**	2	4

12.	**Program definition** Dead Code Elimination in Intermediate Code **Objective** Develop a program that analyzes a given sequence of statements and removes those statements whose results are never used, thereby eliminating unnecessary code and improving execution efficiency. **Language Constraint** The program may be implemented in any programming language. **Input requirement** A list of assignment statements, where each statement may contain: • **Variables** (e.g., x, y, temp1) • **Constants** (e.g., 5, 10) • **Arithmetic operators** `+`, `-`, `*`, `/` • **Unused temporary variables** (candidates for elimination) **Expected output** Display the optimized code after removing: • Statements whose assigned variable is **never used** later • Redundant temporary variable computations **Sample input output** **Input Code** **a = 5** **b = 10** **c = a + b** **d = 7** **e = c * 2** **Output** **a = 5** **b = 10** **c = a + b** **e = c * 2** (Removed: `d = 7` because `d` is never used.) **Test cases -1** **Input** **x = 3** **y = x + 5**	2	5

	z = 10 **w = y * 2** **Out Put** **x = 3** **y = x + 5** **w = y * 2**		