VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

BCSE307L - COMPILER DESIGN

Slot : A2+TA2

1.	**You are given the following statement in a C-like language:** ***total = (price * quantity) + discount > threshold? bonus: 0;*** **Explain how a modern compiler would process this statement through all the phases of compilation, from source code to machine code**

total = (price * quantity) + discount > threshold ? bonus : 0;

This statement uses a **ternary conditional operator**, and a modern compiler would handle it through several well-defined phases. Here's a breakdown of each phase:

1. Lexical Analysis (Scanning)

The compiler reads the raw source code and breaks it into **tokens**, which are the smallest meaningful units.

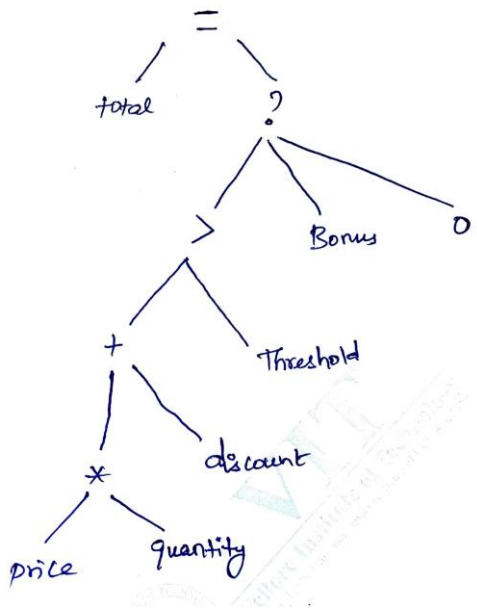
- Tokens identified:
  - total, =, (, price, *, quantity,), +, discount, >, threshold, ?, bonus, :, 0, ;
- Each token is classified (e.g., identifier, operator, literal, punctuation).

2. Syntax Analysis (Parsing)

The compiler builds a **parse tree** or **abstract syntax tree (AST)** to represent the grammatical structure.

- Recognizes the ternary expression:
- ((price * quantity) + discount > threshold) ? bonus : 0
- The AST might look like:

(1) Parse Tree



3. Semantic Analysis

The compiler checks for **type correctness**, **variable declarations**, and **scope rules**.

- Ensures:
    - price, quantity, discount, threshold, bonus, and total are declared.
    - Types are compatible (e.g., all numeric).
    - The ternary operator returns a value that can be assigned to total.

4. Intermediate Code Generation

The compiler translates the AST into an **intermediate representation (IR)**, such as three-address code.

Example IR:

t1 = price * quantity

t2 = t1 + discount

t3 = t2 > threshold

t4 = t3 ? bonus : 0

total = t4

This IR is easier to optimize and target across multiple architectures.

5. Optimization

The compiler applies **optimizations** to improve performance or reduce resource usage.

- **Constant folding**: If any values are known at compile time.
- **Dead code elimination**: If total is never used.
- **Strength reduction**: Replace expensive operations with cheaper ones if possible.

6. Code Generation

The compiler converts IR into **assembly or machine code** for the target architecture.

Example (simplified x86-like pseudocode):

MOV R1, price

MUL R1, quantity

ADD R1, discount

CMP R1, threshold

JLE ELSE_LABEL

MOV R2, bonus

JMP END_LABEL

ELSE_LABEL:

MOV R2, 0

END_LABEL:

MOV total, R2

Each phase transforms the code from human-readable to machine-executable:

Phase	**Purpose**
Lexical Analysis	Tokenize source code
Syntax Analysis	Build AST from grammar
Semantic Analysis	Check types and declarations
Intermediate Code	Generate portable IR
Optimization	Improve performance
Code Generation	Emit target-specific machine code
Linking	Combine and finalize executable

2.	**a) Construct a regular expression for the Language L = {w ∈ {a, b} | *w* starts with optional single occurrence of 'a', followed by zero or more occurrences of "ab", and ends with optional single occurrence of 'b'}**
	b) Identify first_pos, last_pos and follow_pos for the above regular expression using direct method.

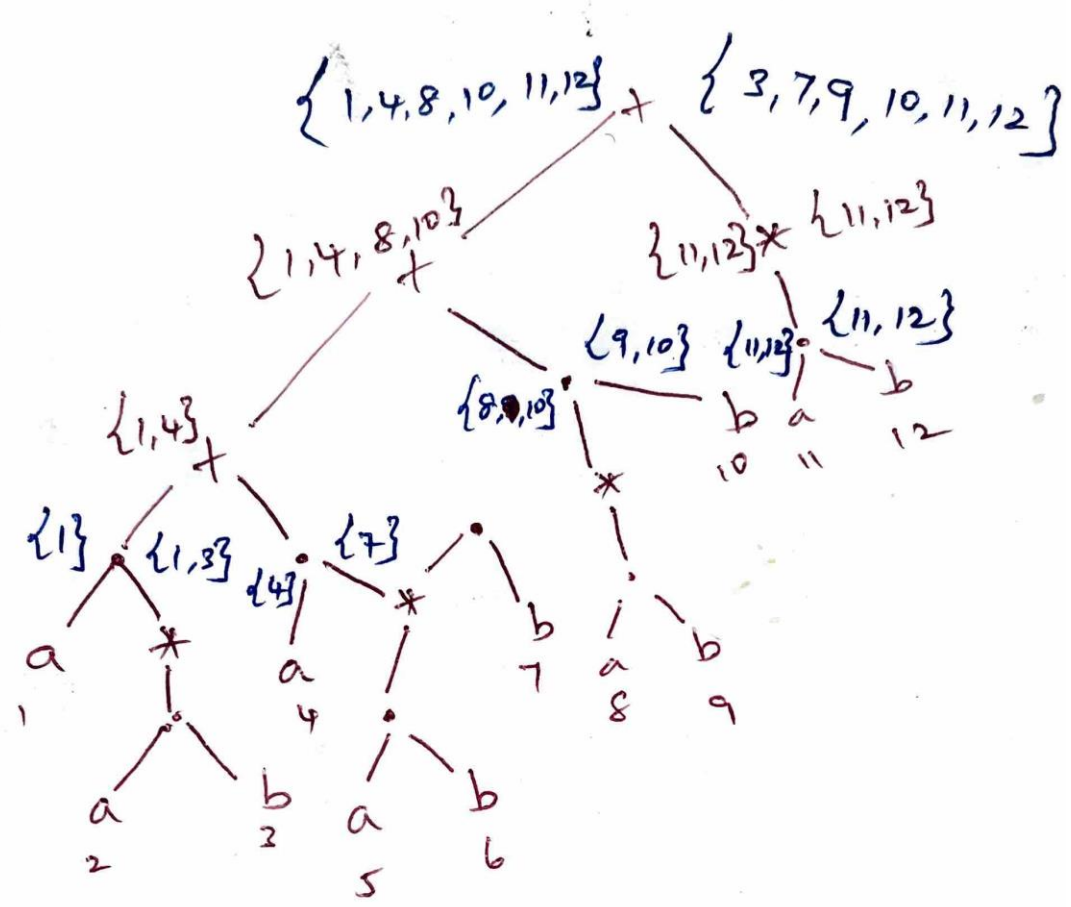
a) There will be two forms of regular expression. Any form is accepted.

a(ab)*+a(ab)*b+(ab)**b*+*(ab)** is equivalent to a?(ab)*b?

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

(2)



Pos	Follow position
1)	{2}
2)	{3}
3)	{2}
4)	{5}
5)	{6}

Pos	Follow pos
6)	{5,7}
7)	—
8)	{9}
9)	{8,10}
10)	—
11)	{12}
12)	{11}

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

3.	**Give a leftmost derivation, a rightmost derivation and parse tree for the following grammars and strings. Argue if the grammar is ambiguous or unambiguous, without resorting to building the parse table. (5 + 5 marks)** **(i) S → S + S|SS|(S)|S □ |a with string (a + a) □ a** **(ii) S → S{S}S | ε with the string {{}{}}**

i) Left most derivation

S

→ S * (using S → S *)

→ (S) * (using S → (S))

→ (S + S) * (using S → S + S inside parentheses)

→ (a + S) * (leftmost S → a)

→ (a + a) * (next S → a)

→ (a + a) * a (final S → a)

Right most derivation

S

→ S * (using S → S *)

→ S * a (final S → a)

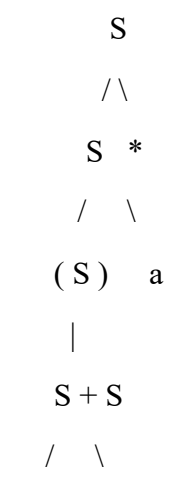
→ (S) * a (S → (S))

→ (S + S) * a (S → S + S inside parentheses)

→ (S + a) * a (rightmost S → a)

→ (a + a) * a (remaining S → a)

Parse Tree:



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

a a

Let's test if the grammar is ambiguous — i.e., whether the same string can be derived in multiple distinct parse trees.

Consider the string: a + a * a

This string could be parsed as:

- a + (a * a) → if * has higher precedence

- (a + a) * a → if + has higher precedence

Since the grammar does not enforce precedence or associativity, both parse trees are valid. That means:

The grammar is ambiguous.

ii)

left most derivation:

S

→ S { S } S

→ ε { S } S (leftmost S → ε)

→ ε { S { S } S } S (S → S { S } S inside braces)

→ ε { ε { S } S } S (leftmost S → ε)

→ ε { ε { ε } S } S (next S → ε)

→ ε { ε { ε } ε } S (next S → ε)

→ ε { ε { ε } ε } ε (final S → ε)

Right most derivation

S

→ S { S } S

→ S { S } ε (rightmost S → ε)

→ S { S { S } S } ε (middle S → S { S } S)

→ S { S { ε } S } ε (inner S → ε)

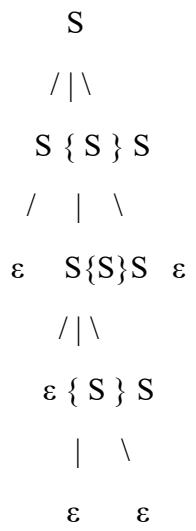
→ S { S { ε } ε } ε (next S → ε)

→ S { ε { ε } ε } ε (leftmost S → ε)

→ ε { ε { ε } ε } ε (final S → ε)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

Parse tree:



In this case, no. The grammar enforces a unique structure for each set of balanced braces. There's only one way to nest and concatenate {{}{}} using the rule S → S{S}S.

The grammar is unambiguous.

4.	**a) Apply suitable transformation for the following grammar, so that it becomes suitable for predictive parsing.** **S → Sa | Sb | cAd | cAe** **A → f | g**
	b) Illustrate whether shift reduce parsing is possible for the string: aaa*a++ on the grammar: S -> SS^{+} | SS* | a

a)

To make the grammar suitable for **predictive parsing**, we need to eliminate **left recursion** and perform **left factoring** where necessary. Predictive parsers require grammars to be **LL(1)** — meaning no left recursion and no ambiguity in choosing productions based on the next input symbol.

Step 1: Eliminate Left Recursion

The productions S → Sa | Sb are **left-recursive** because they start with S. We'll refactor them using the standard transformation:

S → cAd S' | cAe S'

S' → a S' | b S' | ε

Step 2: Left Factoring (if needed)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

Let's look at A → f | g — no left recursion, and no common prefix. So **no transformation needed**.

However, in S → cAd S' | cAe S', both start with cA, so we can **left factor**:

S → cA X

X → d S' | e S'

S → cA X

X → d S' | e S'

S' → a S' | b S' | ε

A → f | g

b)

We'll track the **stack**, **remaining input**, and **action**.

Initial Input: a a a * a + +

Stack	Input	Action
	a a a * a + +	shift
a	a a * a + +	reduce (S → a)
S	a a * a + +	shift
S a	a * a + +	reduce (S → a)
S S	* a + +	shift
S S *	a + +	reduce (S → SS*)
S	a + +	shift
S a	+ +	reduce (S → a)
S S	+ +	shift
S S +	+	reduce (S → SS+)
S	+	shift
S +		✕ cannot reduce

At this point, we're stuck. The stack has S +, and there's no rule that reduces S + directly.
We need **two S symbols followed by +** to reduce via S → SS+.

5.	**Construct the SLR sets of items for the grammar S→SS+|SS*|a with the parsing table. Is the grammar being suitable for SLR?**

S → SS+ | SS* | a

Step 1: Augment the Grammar

We add a new start symbol:

S' → S

S → SS+ | SS* | a

Step 2: Construct Canonical LR(0) Items

We build **item sets** using closure and goto operations.

I_0: Start State

S' → •S

S → •SS+

S → •SS*

S → •a

Closure includes all productions of S since •S is present.

GOTO(I_0, S) → I_1

S' → S•

S → S•S+

S → S•S*

GOTO(I_0, a) → I_2

S → a•

GOTO(I_1, S) → I_3

S → SS•+

S → SS•*

GOTO(I_3, +) → I_4

S → SS+•

GOTO(I_3, *) → I_5

S → SS*•

Step 3: Canonical Collection of LR(0) Items

State	Items
I_0	S' → •S, S → •SS+, S → •SS*, S → •a
I_1	S' → S•, S → S•S+, S → S•S*
I_2	S → a•
I_3	S → SS•+, S → SS•*
I_4	S → SS+•
I_5	S → SS*•

Step 4: FOLLOW(S)

To build the SLR table, we need FOLLOW sets.

FIRST(S)

- From S → a, FIRST(S) = { a }

FOLLOW(S)

- From S' → S, FOLLOW(S') = { $ } → FOLLOW(S) includes $
- From S → SS+, FOLLOW(S) includes +
- From S → SS*, FOLLOW(S) includes *
- So:
- FOLLOW(S) = { +, *, $ }

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST – I – Answer Key
FALL SEMESTER 2025-2026

Step 5: Build the SLR Parsing Table

State	a	+	*	$	S	Action
I_0	s2				1	
I_1	s2			acc	3	
I_2	r3	r3	r3	r3		S → a
I_3	s2	s4	s5			
I_4	r1	r1	r1	r1		S → SS+
I_5	r2	r2	r2	r2		S → SS*

Legend:

- sX = shift and go to state X
- rY = reduce using production Y
- acc = accept

Step 6: Is the Grammar Suitable for SLR?

Yes — the parsing table has:

- **No shift-reduce conflicts**
- **No reduce-reduce conflicts**

Despite the grammar being **ambiguous** (e.g., multiple parse trees for aa+a*), the SLR parser resolves conflicts using FOLLOW sets and **still works**.

The grammar is suitable for SLR parsing.