9	a)	Demonstrate the functioning of Shift-Reduce parsing with a neat diagram and an example.	L3	CO3	5 M
	b)	Write SDT syntax tree for an arithmetic expression a = b+c*d.	L2	CO3	5 M
		UNIT-V			
10	a)	Define Three address code. Outline various Three Address code representations for the expression. x + - y * (- y + z).	L2	CO4	5 M
	b)	Write a short note on i) Type Checking. ii) Type Conversions.	L2	CO4	5 M
		OR			
11	a)	Describe various issues in the design of a Code Generator.	L2	CO4	5 M
	b)	Elaborate Peephole optimization technique.	L3	CO4	5 M

PVP 23

Code: 23IT3503

III B.Tech - I Semester - Regular Examinations - NOVEMBER 2025

AUTOMATA THEORY & COMPILER DESIGN
(INFORMATION TECHNOLOGY)

Duration: 3 hours Max. Marks: 70

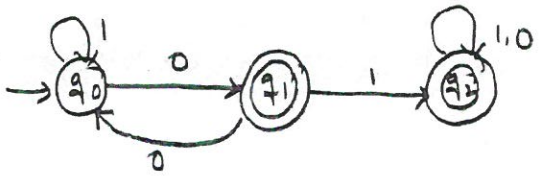
Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

PART – A

		BL	CO
1.a)	What are the applications of Finite Automata?	L2	CO1
1.b)	What are the differences between DFA and NFA?	L2	CO1
1.c)	Describe ambiguous grammar with an example.	L2	CO2
1.d)	Describe halting problem of a Turing Machine.	L2	CO2
1.e)	Differentiate between pass and phase of a compiler.	L2	CO3
1.f)	Define pumping lemma for CFG.	L2	CO2
1.g)	Bottom-up parsing is more powerful than Top-down parsing. Justify.	L2	CO3
1.h)	What do you mean by the strength of attribute grammar?	L2	CO3
1.i)	Explain about Constant folding.	L2	CO4
1.j)	Distinguish between Machine Dependent and Independent optimization.	L2	CO4

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	Define and describe Finite Automata. Illustrate NFA with an example.	L3	CO1	5 M
	b)	Construct DFA to accept strings of a's and b's having an even number of a's and even number of b's.	L3	CO1	5 M
		OR			
3	a)	Construct Finite Automata for the regular expression (11+0)*(00+1)*.	L3	CO1	5 M
	b)	Find the regular expression generated by the following Finite Automata. 	L3	CO2	5 M
		UNIT-II			
4	a)	Design a PDA for equal number of a's and b's.	L3	CO2	5 M
	b)	What is ambiguous grammar? Check whether the given grammar is ambiguous or not? S→ i C t S | i C t S e S | a, C→ b	L3	CO2	5 M
		OR			
5	a)	Describe the process of designing a PDA with given CFG with suitable example?	L2	CO2	5 M

	b)	Construct Turing Machine for accepting strings of the language defined as L= {WCW^r | W ∈ (0 + 1)*}.	L3	CO2	5 M
		UNIT-III			
6	a)	Illustrate different phases of a Compiler for position = initial +rate *60.	L3	CO3	5 M
	b)	Consider the following Conditional statement: if (x > 3) then y = 5 else y = 10; How does a lexical analyser help the above statement in the process of compilation?	L3	CO3	5 M
		OR			
7	a)	State whether the following grammar is LL(1) or not. Justify the answer. S → aBCd | dCBe B → bB | ε C → ca | ac | ε	L3	CO3	5 M
	b)	Write the stepwise procedure of FIRST( ) and FOLLOW( ) with an example.	L3	CO3	5 M
		UNIT-IV			
8		Validate whether the following grammar is SLR or not? Justify. E → E – T | T T → F | *f F → i | (E)	L3	CO3	10M
		OR			

AUTOMATA THEORY &COMPILER DESIGN SCHEME

(INFORMATIONTECHNOLOGY)

PART-A

1.a. What are the applications of Finite Automata? (L2, CO1)
Any two applications----------2M

1.b. What are the differences between DFA and NFA? (L2, CO1)
Any two Differences----------2M

1.c. Describe ambiguous grammar with an example. (L2, CO2)
Definition----------1M
Example---------- 1M

1.d. Describe halting problem of a Turing Machine. (L2, CO2)
Definition----------2M

1.e. Differentiate between pass and phase of a compiler. (L2, CO3)
Any two Differences----------2M

1.f. Define pumping lemma for CFG. (L2,CO2)
Definition----------2M

1.g. Bottom–up parsing is more powerful than Top–down parsing. Justify. (L2,CO3)
Any justification----------2M

1.h. What do you mean by the strength of attribute grammar? (L2,CO3)
Any two strengths----------2M

1.i. Explain about Constant folding. (L2,CO4)
Any two points ----------2M

1.j. Distinguish between Machine Dependent and Machine Independent optimization. (L2,CO4)
Any two Differences----------2M

PART-B

UNIT-1

2a. Define and describe Finite Automata. Illustrate NFA with an example. (L3,CO1,5M)
Definition----------2M
Explanation ----------3M

2b. Construct DFA to accept strings of a's and b's having an even number of a's and even number of b's (L3,CO1,5M)
Machine representation----------1M
Transion Fuction----------1M
Transion Table----------1M
Transition Diagram----------1M
Acceptance----------1M

3a. Construct Finite Automata for the regular expression (11+0)*(00+1)*. (L3,CO2,5M)

Any one approach ----------5M

(i) through NFA with epsilon

(ii) through NFA

3.b. Find the regular expression generated by the given Finite Automata . (L3,CO2,5M)

By applying Aden's theorem generating regular expression----------5M

UNIT-II

4 a. Design a PDA for equal number of a's and b's

(L3,CO2,5M)

Construction of PDA by step by step----------5M

4.b. What is ambiguous grammar? Check whether the given grammar is ambiguous or not.

S → i C t S | i C t S e S | a

C → b (L3,CO2,5M)

Definition----------1M

Checking Through Parse trees or derivative trees ----------4M

5 a Describe the process of designing a PDA with given CFG with suitable example. (L2,CO2,5M)

The process of designing a PDA with given CFG ----------3M

With example----------2M

5b. Construct Turing Machine for accepting strings of the language defined as

L = {WCW' | W □ (0 + 1)*}. (L3 CO2 5M)

Construct Turing Machine ----------5M

UNIT-III

6a. Illustrate different phases of a Compiler for position = initial + rate * 60. (L3 CO3 5M)

Step by step explanation----------5M

6.b. Consider the following Conditional statement:

if (x > 3) then y = 5 else y = 10;

How does a lexical analyser help the above statement in the process of compilation? (L3 CO3 5M)

Procedure of evaluating given statement through lexical analyser----------5M

7.a State whether the following grammar is LL(1) or not. Justify the answer.

S → aBCd | dCBe

B → bB | ε

C → ca | ac | ε (L3 CO3 5M)

Procedure to check given grammar is LL(1) or not through given grammar----------5M

7b. Write the stepwise procedure of FIRST( ) and FOLLOW( ) with an example. (L3 CO3 5M)

The stepwise procedure of FIRST( ) and FOLLOW( ) with an example----------5M

UNIT-IV

8.Validate whether the following grammar is SLR or not? Justify.

E → E - T | T

T → F | *f

F → i | (E) (L3 CO3 10M)

Clear explanation of SLR grammar through given example----------10M

9 a. Demonstrate the functioning of Shift–Reduce parsing with a neat diagram and an example.

(L3 CO3 5M)

Explanation the functioning of Shift–Reduce parsing with a neat diagram and an example -----5M

9.b Write SDT syntax tree for an arithmetic expression:

a = b + c * d. (L2 CO3 5M)

Construction of SDT for given example ----------5M

UNIT-V

10.a Define Three address code. Outline various Three Address code representations for the expression:
x + - y * (- y + z). **(L2 CO4 5M)**
Definition----------1M
Explanation----------4M

10. b) Write a short note on:
i) Type Checking.
ii) Type Conversions. **(L2 CO4 5M)**
Explanation of Type Checking----------2M
Explanation of Type Conversion----------3M

11.a Describe various issues in the design of a Code Generator. (L2 CO4 5M)
Explanation of various issues in the design of a Code Generator ----------5M

11.b Elaborate Peephole optimization technique. **(L3 CO4 5M)**
Definition----------1M
Explanation----------2M

AUTOMATA THEORY &COMPILER DESIGN SCHEME

(INFORMATIONTECHNOLOGY)

PART-A

1.a. What are the applications of Finite Automata? (L2, CO1)

- Lexical Analysis in compilers (token recognition).
- Text search algorithms (e.g., pattern matching).
- String validation (email, identifiers, passwords).
- Protocol design in networking.
- Control systems such as vending machines.

1.b. What are the differences between DFA and NFA? (L2, CO1)

DFA	NFA
Only one transition for each symbol from a state	Can have multiple transitions for same symbol
No ε-moves	Allows ε-moves
Unique next state	Multiple possible next states
Easy to implement	Easier to design, but harder to implement
Always deterministic	Non-deterministic behaviour

1.c. Describe ambiguous grammar with an example. (L2, CO2)

A grammar is ambiguous if a single string has more than one parse tree.
For example
$E \rightarrow E + E \mid E * E \mid id$
String: id + id * id
This grammar produces **two parse trees** because + and * precedence is not enforced.

1.d. Describe halting problem of a Turing Machine. (L2, CO2)

The Halting Problem is determining whether a computer program will eventually stop or run forever. Creating a general algorithm that can accurately predict this for all programs is impossible. Alan Turing's proof showed no way to solve the Halting Problem for all cases. Halting means that the system will either accept or halt(reject) a certain input to avoid entering an infinite loop. The decision of whether or not a specific Turning machine should halt is known as the Halting Problem of Turning Machines.
The halting problem is one of the most well-known problems proven to be undecidable

1.e. Differentiate between pass and phase of a compiler. (L2, CO3)

Pass	Phase
A complete traversal of the program	Logical subdivision of compilation
Compiler may have 1-pass or 2-pass	Many phases (lexical, syntax, semantic…)
Uses intermediate files	Does not depend on passes

1.f. Define pumping lemma for CFG. (L2,CO2)

The pumping lemma is used to prove that a language is not context-free.
Statement:
For every context-free language L, there exists a constant p (pumping length) such that any string
$s \in L$ with length $\geq p$ can be written as:
$s = uvxyz$
where:

1. $|vxy| \leq p$
2. $|vy| \geq 1$
3. For all $k \geq 0$, uv^k x y^k z ∈ L

1.g. Bottom–up parsing is more powerful than Top–down parsing. Justify. (L2,CO3)

- Bottom-up parsers (LR parsers) can handle a larger class of grammars, including left-recursive grammars.
- Top-down parsers (LL) cannot handle:
- Left recursion
- Many ambiguous constructs
- LR parsers detect errors as soon as possible and are more powerful for programming languages.

1.h. What do you mean by the strength of attribute grammar? (L2,CO3)

The strength of an attribute grammar refers to how effectively it can specify semantic information such as:

- Type checking
- Scope rules
- Intermediate code generation

A stronger attribute grammar can express more semantic constraints directly within the grammar.

1.i. Explain about Constant folding. (L2,CO4)

Constant folding is a compile-time optimization where constant expressions are evaluated before runtime.

Example:

int x = 5 + 3;

Compiler replaces it with:

int x = 8;

This reduces runtime computation and improves performance.

1.j. Distinguish between Machine Dependent and Machine Independent optimization.

Machine Dependent	Machine Independent
Depends on target architecture	Works for any machine
Register allocation, instruction selection	Dead-code elimination, constant folding
Focus on hardware usage	Focus on improving IR
CPU pipeline aware	Purely logical optimizations

PART-B

UNIT-1

2a. Define and describe Finite Automata. Illustrate NFA with an example. (L3,CO1,5M)

ANSWER:

A Finite Automaton (FA) is a mathematical model of computation used to recognize patterns or regular languages.

It consists of a finite number of states and processes an input string symbol by symbol to determine whether the string is accepted.

A Finite Automaton is represented as a 5-tuple:

M = (Q, Σ, δ, q_0, F) where

- **Q** = finite set of states
- **Σ** = finite input alphabet
- **δ** = transition function
- **q_0** = initial state
- **F** = set of accepting states

There are two types:

1. **Deterministic Finite Automaton (DFA)**
2. **Non-Deterministic Finite Automaton (NFA)**

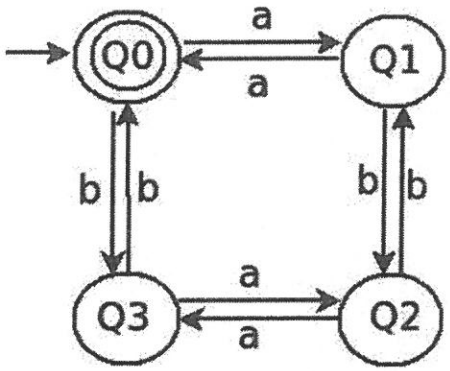
NFA Example

Let NFA accept strings ending with **1**.
Q = {q0, q1}
Σ = {0,1}
Transitions:

- δ(q0, 0) = {q0}
- δ(q0, 1) = {q0, q1}
- δ(q1, 0) = Ø
- δ(q1, 1) = Ø

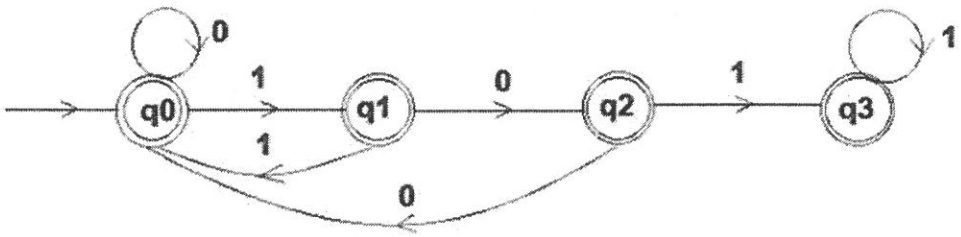
q_0 = q0
F = {q1}
This NFA moves to q1 when last symbol is 1 → accepted.

2b. Construct DFA to accept strings of a's and b's having an even number of a's and even number of b's
(L3,CO1,5M)
ANSWER:

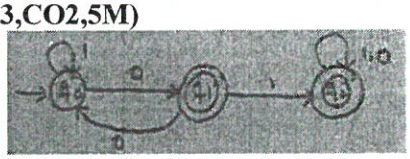


3a. Construct Finite Automata for the regular expression (11+0)*(00+1)*.
(L3,CO2,5M)
ANSWER:

(i) through NFA with epsilon
(ii) through NFA



3.b. Find the regular expression generated by the given Finite Automata .
(L3,CO2,5M)



ANSWER:

Ans By applying adan's theorem;

$q_0 = 1q_0 + 0q_1 + \epsilon$ — ①

$q_1 = 0q_0 +$ — ②

$q_2 = 1q_1 + 0q_2 + 1q_2$ — ③

for regular Expression evaluating q_1 and q_2.

replace q_1 in equation ①

$q_0 = 1q_0 + 0(1q_0) + \epsilon$

$\Rightarrow = 1q_0 + 01q_0 + \epsilon$

$\Rightarrow q_0(1+01) + \epsilon$

$q_0 = q_0(1+01) + \epsilon$

By applying adan's theorem

$R = Q + RP$ then solution is $R = QP^*$

$q_0 = (1+01)^*$

replace q_0 value in equation ②.

$q_1 = 0(1+01)^*$

By evaluating equation ③

$q_2 = 10(1+01)^* + q_2(0+1)$

After applying adan's theorem.

$q_2 = 10(1+01)^*(0+1)^*$

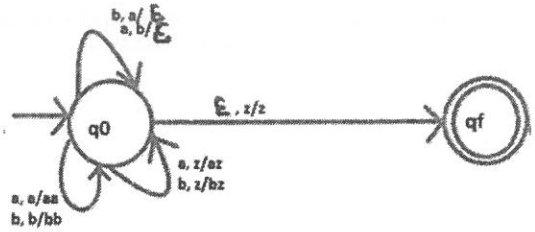
for final R.E Union of q_1 & q_2 is.

$R.E = 0(1+01)^* + 10(1+01)^*(0+1)^*$

UNIT-II

4 a. Design a PDA for equal number of a's and b's
(L3,CO2,5M)
ANSWER:

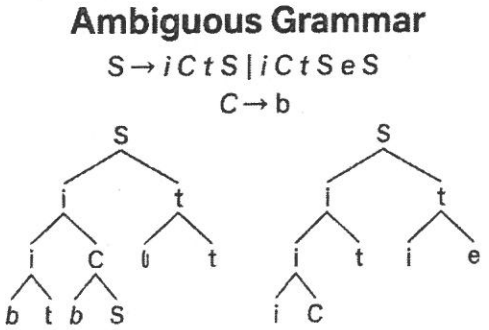
L = {ab, aabb, abba, aababb, bbabaa, baaababb, ...}
(q0, a, z) ⊢(q0, az)
δ(q0, a, a) ⊢(q0, aa)
δ(q0, b, z) ⊢(q0, bz)
δ(q0, b, b) ⊢(q0, bb)
δ(q0, a, b) ⊢(q0, ϵ)
δ(q0, b, a) ⊢(q0, ϵ)
δ(q0, ϵ, z) ⊢(qf, z)
Where, q0 = Initial state , qf = Final state
ϵ = indicates pop operation



1.Scan string from left to right.
2.on input 'a' and STACK alphabet Z, push the 'a's into STACK as : (a,Z/aZ) and state will be q0.
3.second input 'a' and STACK alphabet 'a', push the 'a's into STACK as : (a,a/aa) and state will be q0.
4.Third input 'b' and STACK alphabet 'a', pop from STACK as : (b,a/∈) and state will be q0.
5.on input 'b' and STACK alphabet 'a', pop from STACK as : (b,a/∈) and state will be q0.
6.on input 'b'and STACK alphabet Z, push the 'b's into STACK as : (b,Z/bZ) and state will be q0.
7.on input 'a' and STACK alphabet 'b', pop from STACK as : (a,b/∈) and state will be q0.
8.on input ∈ and STACK alphabet Z, go to final state(qf) as : (∈, Z/Z).
So, at the end the stack becomes empty then we can say that the string is accepted by the PDA.

4.b. What is ambiguous grammar? Check whether the given grammar is ambiguous or not.
S → i C t S | i C t S e S | a
C → b
(L3,CO2,5M)
ANSWER:

A grammar is ambiguous if a string can have more than one parse tree or leftmost derivation.
Given Grammar
S → i C t S | i C t S e S | a
C → b
String: i b t i b t a e a



This produces two different parse trees (attach else to first or second if).
Hence, Grammar is ambiguous.

5 a Describe the process of designing a PDA with given CFG with suitable example.
(L2,CO2,5M)
ANSWER:

Start the PDA with a Single Start Configuration

- Create a start state q_0.
- Push the start variable S of the CFG onto the stack.
- The input head is at the beginning of the input string.

Transition:

$$(q_0, \epsilon, \epsilon) \rightarrow (q_0, S)$$

Apply Production Rules Using Stack Substitution
For every production in the CFG:

$$A \rightarrow \alpha$$

Create a PDA transition:

$$(q, \epsilon, A) \rightarrow (q, \alpha)$$

Meaning:

- If the top of the stack is variable A,
- Replace it by the RHS α (push α in reverse order).

This simulates grammar derivation on the stack.

Step 3: Match Terminals with Input Symbols
Whenever the top of the stack is a terminal a,
Consume an a from the input.
Transition:

$$(q, a, a) \rightarrow (q, \epsilon)$$

This removes the matching terminal from the stack and consumes it from input.

Step 4: Accept the String
The PDA accepts when:

- The entire input is processed, and
- The stack becomes empty.

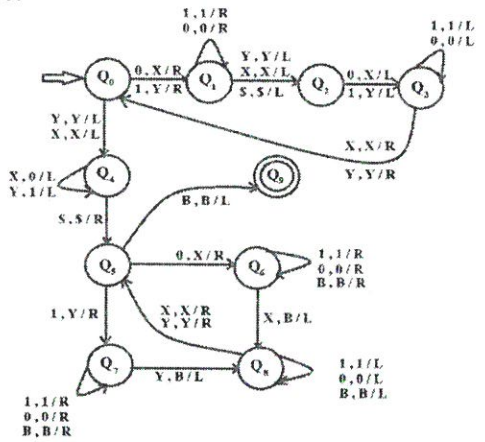
Transition:

$$(q, \epsilon, \epsilon) \rightarrow (q_{accept}, \epsilon)$$

5b. Construct Turing Machine for accepting strings of the language defined as L = {WCW' | W □ (0 + 1)*}. (L3 CO2 5M)

ANSWER:

Formal 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
where:
$\Sigma = \{0,1,C\}$(input alphabet),
$\Gamma = \{0,1,C,X,Y,B\}$(tape alphabet),
$q_0$is the start state,
$B$is the blank symbol,
$F = \{q_{acc}\}$is the accepting state.



UNIT-III

6a. Illustrate different phases of a Compiler for position = initial + rate * 60. (L3 CO3 5M)

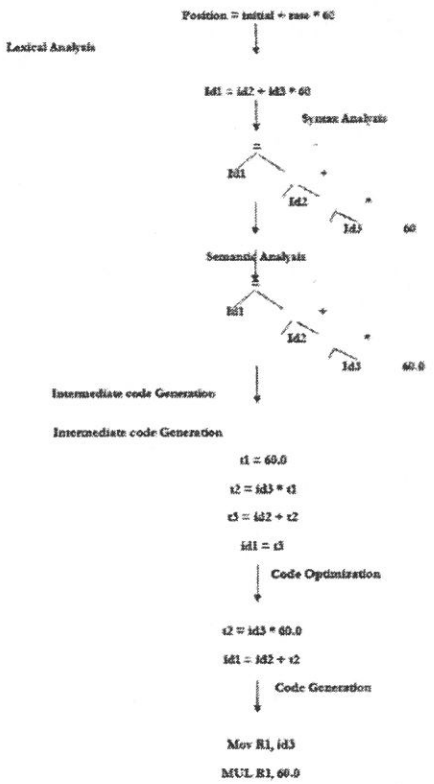
ANSWER:

A compiler is a software tool that translates high-level programming languages into machine code that can be executed by a computer. It typically consists of several phases, each performing a specific task in the compilation process. The main phases of a compiler are:

1. Lexical Analysis: This phase scans the source code and breaks it into a sequence of tokens. Tokens are the smallest meaningful units of the programming language, such as keywords,

identifiers, operators, and literals. For example, in the expression position: = initial + rate * 60, the tokens would be position, :, =, initial, +, rate, *, and 60.

2. Syntax Analysis (Parsing): In this phase, the tokens produced by the lexical analysis phase are organized into a hierarchical structure called a parse tree or abstract syntax tree (AST). The parse tree represents the syntactic structure of the program. It checks if the program follows the grammar rules of the programming language. For the given expression, the parse tree would represent the assignment of the sum of initial and the product of rate and 60 to the variable position.
3. Semantic Analysis: This phase checks the meaning of the program by analyzing the parse tree. It ensures that the program follows the language's semantics and performs type checking. It also resolves variable references and enforces language-specific rules. For example, it would check if the variables initial, rate, and position are declared and have compatible types.
4. Intermediate Code Generation: In this phase, the compiler generates an intermediate representation of the program. This representation is usually closer to the machine code but still independent of the target machine. It simplifies the subsequent optimization and code generation phases. The intermediate code may be in the form of three-address code, quadruples, or an abstract stack machine.
5. Code Optimization: This phase improves the intermediate code by applying various optimization techniques. It aims to make the program more efficient in terms of execution time and memory usage. Common optimizations include constant folding, common subexpression elimination, loop optimization, and register allocation.
6. Code Generation: In the final phase, the compiler translates the optimized intermediate code into the target machine code. It maps the high-level language constructs to the corresponding machine instructions. The generated code is specific to the target architecture and can be executed directly by the computer.



6.b. Consider the following Conditional statement:
if (x > 3) then y = 5 else y = 10;
How does a lexical analyser help the above statement in the process of compilation? (L3 CO3 5M)
ANSWER:

A **Lexical Analyzer (Scanner)** is the first phase of a compiler. Its main job is to read the source program character by character and convert it into a sequence of **tokens**. Tokens are meaningful lexical units such as keywords, identifiers, operators, constants, and punctuation symbols.
The lexical analyzer helps the above conditional statement in the following ways:
Breaks the Statement into Tokens
The given statement is divided into the following tokens:

Lexeme Token Type

If	KEYWORD
(	LEFT_PAREN
X	IDENTIFIER
>	RELATIONAL_OPERATOR
3	CONSTANT
)	RIGHT_PAREN
then	KEYWORD
Y	IDENTIFIER
=	ASSIGNMENT_OPERATOR
5	CONSTANT
else	KEYWORD
Y	IDENTIFIER
=	ASSIGNMENT_OPERATOR
10	CONSTANT
;	SEMICOLON

The lexical analyzer converts the raw characters into these structured tokens.

Assigns Attributes to Tokens

For identifiers and constants, the lexical analyzer creates symbol table entries:

- Identifier **x** → variable type (later verified), memory location
- Identifier **y** → symbol table entry
- Constant **3**, **5**, **10** → stored as literal constants

Each token is returned along with its attribute (pointer to symbol table entry).

Detects Lexical Errors

If the programmer writes something invalid like:
y = @5;
The lexical analyzer detects the invalid symbol @ and reports an error **before syntax analysis**.
For the given statement, if everything is correct, no lexical errors are reported.

Supplies Tokens to the Syntax Analyzer

The lexical analyzer sends a token stream to the parser:
IF (ID > NUM) THEN ID = NUM ELSE ID = NUM ;
This allows the syntax analyzer to construct a parse tree for the conditional statement.

7.a State whether the following grammar is LL(1) or not. Justify the answer.
S → aBCd | dCBe
B → bB | ε
C → ca | ac | ε **(L3 CO3 5M)**

ANSWER:

Given grammar:
S → aBCd | dCBe
B → bB | ε
C → ca | ac | ε
To check whether a grammar is **LL(1)**, we apply the following conditions:
A grammar is LL(1) if:

1. **No FIRST/FIRST conflicts**
2. **No FIRST/FOLLOW conflicts**
3. **No left recursion or ambiguity**

We must compute **FIRST** and **FOLLOW** sets and check the above conditions.

Step 1 – FIRST Sets

FIRST(B):
B → bB | ε
FIRST(B) = { b, ε }
FIRST(C):
C → ca | ac | ε
FIRST(C) = { c, a, ε }
FIRST(S):
S → aBCd | dCBe
FIRST(aBCd) = a
FIRST(dCBe) = d

FIRST(S) = { a, d }
No conflict here because the two alternatives of **S** start with **distinct terminals (a, d)**.

Step 2 – FOLLOW Sets
FOLLOW(S):
S is the start symbol →
FOLLOW(S) = { $ }
FOLLOW(B):
From **S → aBCd**
B is followed by **C**
→ FIRST(C) − {ε} = { c, a }
From **S → dCBe**
B is followed by terminal **e**
→ FOLLOW(B) = { c, a, e }
And since **B → bB | ε**, ε-production does not add FOLLOW(S).
FOLLOW(C):
From **aBCd**
C is followed by **d**
From **dCBe**
C is followed by **B** → FIRST(B) − {ε} = { b }
Also **B ⇒ ε**, so FOLLOW(C) also gets FOLLOW(S) = { $ }
→ FOLLOW(C) = { d, b, $ }

Step 3 – LL(1) Conditions Checking
Condition 1: FIRST/FIRST conflict
Check alternatives of each non-terminal.

- **S**: FIRST(aBCd) = a, FIRST(dCBe) = d No conflict.
- **B**: FIRST(bB) = b, FIRST(ε) = ε
  → Need to check **FIRST(ε)** vs **FOLLOW(B)**
  FIRST(ε) = { ε }
  FOLLOW(B) = { c, a, e }

No intersection → No conflict.

- **C**: FIRST(ca) = c
  FIRST(ac) = a
  FIRST(ε) = ε
  For ε-production, check FOLLOW(C) = { d, b, $ }

FIRST(ε) ∩ FOLLOW(C) = ∅
No conflict.
Condition 2: FIRST/FOLLOW conflict

- For **B → ε**, FIRST(B) contains ε
  Check FIRST(B) ∩ FOLLOW(B)
- {ε} ∩ {a, c, e} = ∅
- → OK
- For **C → ε**, FIRST(C) contains ε
  Check FIRST(C) ∩ FOLLOW(C)
- {ε} ∩ {d, b, $} = ∅
- → OK

No FIRST/FOLLOW conflicts.

Since the grammar:

- Has **no FIRST/FIRST conflicts**
- Has **no FIRST/FOLLOW conflicts**
- Contains **no left recursion**
- All nullable symbols satisfy LL(1) conditions

The grammar is LL(1).

7b. Write the stepwise procedure of FIRST( ) and FOLLOW( ) with an example. (L3 CO3 5M)
ANSWER:

FIRST( )
FIRST(X) is the set of terminals that can appear as the first symbol of some string derived from the grammar symbol X (terminal or nonterminal). ε is included in FIRST(X) if X can derive the empty string.

Procedure to compute FIRST( )

1. **Initialize:**
   - For every terminal a: FIRST(a) = { a }.
   - For every nonterminal A: start with FIRST(A) = ∅.
2. **For each production A → α do:**
   - Scan α from left to right. For each symbol Y in α:
     - Add FIRST(Y) - {ε} to FIRST(A).
     - If ε ∈ FIRST(Y), continue to the next symbol; otherwise stop for this production.
   - If every symbol in α can derive ε, add ε to FIRST(A).
3. **Iterate** step 2 until no FIRST set changes (fixed point).
   (This algorithm follows the standard rules given in the attached notes.)

FOLLOW( )
FOLLOW(A) is the set of terminals that can appear immediately to the right of nonterminal A in some sentential form. $ (end marker) is placed in FOLLOW(S) for start symbol S.

Procedure to compute FOLLOW( )

1. **Initialize:**
   - For every nonterminal A: FOLLOW(A) = ∅.
   - Put $ in FOLLOW(S) (where S is the start symbol).
2. **For every production A → α B β:**
   - Add FIRST(β) - {ε} to FOLLOW(B).
   - If β can derive ε (i.e., ε ∈ FIRST(β)), add FOLLOW(A) to FOLLOW(B).
3. **For every production A → α B (i.e., B at end):**
   - Add FOLLOW(A) to FOLLOW(B).
4. **Iterate** steps 2–3 until no FOLLOW set changes (fixed point).
   (These rules are the standard ones in your notes.)

Use the common expression grammar after eliminating left recursion:
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Compute FIRST sets

- FIRST(id) = { id }, FIRST('(') = { (} (terminals).
- FIRST(F) = { (, id } (from F → (E) and F → id).
- FIRST(T') = { * , ε }.
- FIRST(T) = FIRST(F) = { (, id }.
- FIRST(E') = { + , ε }.
- FIRST(E) = FIRST(T) = { (, id }.
  (So list them: FIRST(E)={(,id}, FIRST(E')={+,ε}, FIRST(T)={(,id}, FIRST(T')={*,ε}, FIRST(F)={(,id}).

Compute FOLLOW sets

- Start: FOLLOW(E) = { $,) } (because E can appear inside ( ) and it is start symbol $).
- From productions and propagation:
  - FOLLOW(E') = FOLLOW(E) = { $,) }.
  - FOLLOW(T) includes FIRST(E') - {ε} = { + } and also FOLLOW(E) if E' can be ε,

so FOLLOW(T) = { +, $,) }.
- FOLLOW(T') = FOLLOW(T) = { +, $,) }.
- FOLLOW(F) includes FIRST(T') - {ε} = { * } and propagation from FOLLOW(T') when T' can be ε, so FOLLOW(F) = { +, *, $,) }.

Final sets:
FOLLOW(E) = { $,) }, FOLLOW(E') = { $,) }, FOLLOW(T) = { +, $,) }, FOLLOW(T') = { +, $,) }, FOLLOW(F) = { +, *, $,) }.

How to use FIRST & FOLLOW (brief note for exam)

- To build the predictive parsing table M[A, a]: for each production A → α, put A → α in M[A, a] for all a ∈ FIRST(α) - {ε}. If ε ∈ FIRST(α), also put A → α in M[A, b] for every b ∈ FOLLOW(A) (and in M[A,$] if $ ∈ FOLLOW(A)). Repeat until table entries are complete.

UNIT-IV

8.Validate whether the following grammar is SLR or not? Justify.
E → E - T | T
T → F | *f
F → i | (E) **(L3 CO3 10M)**

ANSWER:

(0) E' → E
(1) E → E - T
(2) E → T
(3) T → F
(4) T → * F
(5) F → i
(6) F → (E)

To check SLR(1) we must:

1. Augment the grammar (done above with E' → E).
2. Build the canonical collection of LR(0) items (closures & gotos).
3. From that collection, build the SLR ACTION and GOTO tables:
   - For an item [A → α · a β] with terminal a, ACTION[state, a] = shift to goto(state, a).
   - For an item [A → α ·] (dot at right end), for every terminal a ∈ FOLLOW(A), ACTION[state, a] = reduce A → α.
   - If item is [E' → E ·] then ACTION[state, $] = accept.
4. If any cell in ACTION gets two different actions (shift/reduce or reduce/reduce), the grammar is not SLR. Otherwise it is SLR.

To be SLR we only use LR(0) item sets (no lookahead) and use FOLLOW sets to place reductions.

Compute FIRST (quickly for terminals/nonterminals used):

- FIRST(i) = { i }, FIRST(() = {(}, FIRST()) = {) }, FIRST(*) = { * }.
- FIRST(F) = { i, (}.
- FIRST(T) = FIRST(F) = { i, (} (since *F starts with *, include * also: actually FIRST(T) = { i, (, * } because T → *F).
- FIRST(E) = FIRST(T) = { i, (, * }.

Compute FOLLOW sets (sketch):

- FOLLOW(E') = { $ }.
- FOLLOW(E) contains $ (start symbol) and) (because E can appear inside (E)), so FOLLOW(E) = { $,) }.
- From E → E - T we get FOLLOW(E) is unchanged for left E. For T in E → E - T we have FOLLOW(T) ⊇ FOLLOW(E)? Actually T is at end so FOLLOW(T) contains FOLLOW(E) = { $,) }.
- From T → * F the F is at end, so FOLLOW(F) ⊇ FOLLOW(T).
- Summarizing (after propagation) we get:

- FOLLOW(E) = { $,) }
- FOLLOW(T) = { $,) }
- FOLLOW(F) = { $,) }

We built (conceptually) the canonical LR(0) items and examined the states that could cause conflicts.

All reductions (for `F → i, F → (E), T → F, E → T, E → E - T`) are placed only on terminals in the respective FOLLOW sets (`{ $, ) }`).

Shift actions are needed on terminals `{ i, (, *, - }` in various states; these terminals are not in the FOLLOW sets where the reductions apply.

Therefore the ACTION table has no shift/reduce or reduce/reduce conflicts when reductions are placed using FOLLOW sets (the SLR rule).

The grammar (with the assumption `T → *F`) is SLR.

9 a. Demonstrate the functioning of Shift–Reduce parsing with a neat diagram and an example. (L3 CO3 5M)

ANSWER:

Shift–reduce parsing is a bottom-up parsing method used to reduce an input string to the start symbol by repeatedly applying grammar rules in reverse. It uses a stack and an input buffer. At each step, the parser decides whether to **shift** the next input symbol onto the stack or **reduce** the top of the stack using a production rule. A reduce operation replaces a sequence of symbols on the stack with a nonterminal. The process continues until the entire string is reduced to the start symbol, and if no error occurs, the parser accepts the input. Shift–reduce parsing mainly uses four operations: **SHIFT**, **REDUCE**, **ACCEPT**, and **ERROR**. The parser works step by step, and the choice of shift or reduce is based on the pattern of symbols at the top of the stack.

To demonstrate this, consider the grammar:
E → E + T
E → T
T → id

We parse the input string **id + id**. The table below shows how the parser gradually shifts symbols and reduces them using the grammar rules.

Shift–Reduce Parsing Table

Stack	Input	Action
—	id + id $	SHIFT
id	+ id $	REDUCE (T → id)
T	+ id $	REDUCE (E → T)
E	+ id $	SHIFT
E +	id $	SHIFT
E + id	$	REDUCE (T → id)
E + T	$	REDUCE (E → E + T)
E	$	ACCEPT

9.b Write SDT syntax tree for an arithmetic expression:
a = b + c * d. **(L2 CO3**
5M)
ANSWER:

a = b + c * d

This corresponds to the usual precedence:

- * has higher precedence than +
- assignment = is lowest

SDT Syntax Tree for a = b + c * d
Tree Structure (Text Representation)

```
        (=)
       /   \
    id(a)   (+)
           /   \
        id(b)   (*)
               /   \
            id(c)  id(d)
```

Explanation

- The **root node** is the assignment operator =.
- Left child of = is id(a) (the variable being assigned).
- Right child of = is the expression b + c * d.
- In the right subtree:
  - The root is +.
  - Left child of + is id(b).
  - Right child of + is * because multiplication has higher priority.
  - Under *, left child is id(c) and right child is id(d).

This is the **standard SDT syntax tree** used in compilers to evaluate or generate three-address code.

UNIT-V

10.a Define Three address code. Outline various Three Address code representations for the expression:
x + - y * (- y + z). **(L2 CO4**
5M)
ANSWER:

Three Address Code (TAC) is an intermediate representation used in compilers where each instruction contains at most three addresses (operands).
A TAC statement typically has the form:
x = y op z
where

- x is the target,
- y and z are operands,
- op is a binary or unary operator.

It breaks complex expressions into simple step-by-step operations suitable for code generation and optimization.

TAC for the expression:
Expression:
[
x + -y * (-y + z)

]
We rewrite it clearly:
x = x + (-y * (-y + z))
We will generate TAC by processing innermost expressions first.

Various Three Address Code Representations
1. Normal Three Address Code (Quadruples-like form)
Let us generate temporary variables:
t1 = - y
t2 = - y
t3 = t2 + z
t4 = t1 * t3
t5 = x + t4
x = t5

2. Quadruples Representation

op	arg1	arg2	result
unary-	y	—	t1
unary-	y	—	t2
+	t2	z	t3
*	t1	t3	t4
+	x	t4	t5
=	t5	—	x

3. Triples Representation
(Triples use instruction numbers instead of temporary variable names.)

#	op	arg1	arg2
(0)	unary-	y	—
(1)	unary-	y	—
(2)	+	(1)	z
(3)	*	(0)	(2)
(4)	+	x	(3)
(5)	=	(4)	x

4. Indirect Triples Representation
Either pointer table is used to refer to instruction numbers:
Pointer Table

ptr	refers to
p0	0
p1	1
p2	2
p3	3
p4	4
p5	5

#	op	arg1	arg2
0	unary-	y	—
1	unary-	y	—

#	op	arg1	arg2
2	+	(1)	z
3	*	(0)	(2)
4	+	x	(3)
5	=	(4)	x

10. b) Write a short note on:
i) Type Checking.
ii) Type Conversions. **(L2 CO4**
5M)
ANSWER:

Type checking is the process of verifying that the operands of an operator are **compatible in type** according to the language's rules. It ensures that expressions, assignments, and function calls follow correct type constraints **before execution** (usually during compilation).

- Performed by the **semantic analyzer** of the compiler.
- Uses a **symbol table** to track type information of variables, functions, arrays, structures, etc.
- Detects errors like:
  - Adding an integer to a string
  - Assigning a float value to a boolean
  - Passing wrong-type arguments to functions
- Helps ensure **type safety** and prevents runtime failures.

Advantages

- Early error detection
- Improves program reliability
- Supports compiler optimizations

Type conversion is the process of **converting a value of one data type into another**. Conversions occur when operand types do not match or when a specific type is required.

Types of Type Conversions

1. Implicit Conversion (Coercion)

- Performed **automatically** by the compiler.
- Used when safe conversions are possible.
- Example:
  - int → float
  - char → int

2. Explicit Conversion (Casting)

- Programmer-specified conversion.
- Syntax like:
- float x = (float) y;
- Required for narrowing conversions such as:
  - float → int
  - long → short

Type Conversion Rules

- Usually follow a **hierarchy of widening**:
  char → int → float → double
- When types differ in an expression, the compiler promotes them to a **common type**.

11.a Describe various issues in the design of a Code Generator. L2 CO4 5M

Here is your **expanded, simple 5-mark answer** written in **three short student-like paragraphs**:

ANSWER:

Issues in the Design of a Code Generator

The code generator is the compiler phase that converts intermediate code into machine instructions, and its design faces many challenges. One major issue is understanding the

target machine architecture. Different machines have different numbers of registers, instruction formats, and addressing modes, so the generator must produce code that works correctly on the specific hardware. It should also choose the most suitable machine instructions so that the program runs efficiently.
Another issue is register allocation. Since registers are limited, the code generator must decide which variables or temporary values should stay in registers and which should be placed in memory. If this is not handled properly, the generated code becomes slow because unnecessary load and store operations are added. The generator must also consider how data types like integers, floating-point values, and characters are stored and accessed, and ensure that type conversions and operations are done correctly.
Finally, the code generator must balance **correctness** and **optimization**. The highest priority is to produce code that behaves exactly as the program intends, but at the same time it should try to make the code as efficient as possible by reducing instruction count and improving execution speed. It must also follow runtime environment rules such as stack usage, parameter passing, and return values. All these factors make the design of a code generator a challenging but important task in a compiler.

11.b Elaborate Peephole optimization technique. **(L3 CO4 5M)**

ANSWER:

Peephole optimization is a **local optimization technique** used in the code generation phase of a compiler. Instead of looking at the entire program, the compiler looks at a **small window of instructions**, called a *peephole*, and tries to improve them. This window usually contains **2–5 instructions**, and the goal is to make the generated machine code shorter, faster, or more efficient without changing the meaning of the program.
One common use of peephole optimization is **eliminating redundant instructions**. For example, if the code loads a value into a register and immediately stores it back to the same memory location, the store might be unnecessary. Another improvement is **constant folding**, where operations on constants (like 4 + 5) are replaced by their computed result (9). Peephole optimization also removes **unreachable code** such as jumps that skip to the next instruction, and it simplifies **jumps to jumps**, replacing chains of jumps with a single direct jump.
Peephole optimization can also apply **strength reduction**, where expensive operations are replaced with cheaper ones. For example, multiplication by 2 can be replaced by a left shift operation, which is faster on many machines. Additionally, peephole optimization ensures better register use by replacing instructions that unnecessarily move data between registers. Although peephole optimization works on a very small part of the code at a time, it significantly improves the quality of the final machine code.