3. a) Explain the role of parser. Explain types of grammars used for parsing. [7M]

Role of Parser: The parser is a crucial component in the compiler design process. Its main role is to check the syntax of the source code to ensure it adheres to the grammatical rules of the programming language. The parser takes input from the lexical analyzer in the form of tokens and constructs a parse tree or syntax tree. It helps in detecting syntax errors and provides meaningful error messages.

Types of Grammars Used for Parsing:

1. **Context-Free Grammar (CFG):** Used extensively in syntax analysis, CFG consists of production rules where each rule maps a single non-terminal symbol to a string of non-terminal and/or terminal symbols.
2. **Regular Grammar:** Simplest form, used in lexical analysis, suitable for regular expressions.
3. **Context-Sensitive Grammar:** More powerful than CFG, used in cases where the context of a symbol affects its interpretation.
4. **Unrestricted Grammar:** The most general form, capable of describing any language recognized by a Turing machine, but impractical for parsing.

---

3. b) Write an algorithm for constructing a predictive parsing table. Give Example [7M]

Algorithm for Constructing Predictive Parsing Table:

1. Compute FIRST and FOLLOW sets for the grammar.
2. For each production A → α:
    - For each terminal 'a' in FIRST(α), add A → α to the parsing table entry M[A, a].
    - If ε (epsilon) is in FIRST(α), then for each terminal 'b' in FOLLOW(A), add A → α to M[A, b].
3. If any entry has more than one production, the grammar is not LL(1).

Example: Grammar: S → aABe A → b | ε B → d | ε

- FIRST(S) = {a}, FIRST(A) = {b, ε}, FIRST(B) = {d, ε}
- FOLLOW(S) = {$}, FOLLOW(A) = {d, e}, FOLLOW(B) = {e}

Constructing the table based on FIRST and FOLLOW sets.

(OR)

4. a) What is an ambiguous grammar? Write a procedure to eliminate the same with an example. [7M]

Ambiguous Grammar: A grammar is ambiguous if there exists at least one string that can have more than one distinct parse tree or leftmost/rightmost derivation.

Procedure to Eliminate Ambiguity:

1. Identify the ambiguous productions.
2. Rewrite the grammar to remove multiple parse trees for the same string.
3. Use left factoring and introduce new non-terminals if necessary.

Example: Ambiguous Grammar: E → E + E | E * E | id

Eliminating Ambiguity: E → E + T | T T → T * F | F F → id

---

4. b) Consider the following grammar: S → (L) | a
L → L, S | S

Construct leftmost and rightmost derivations and parse trees for the following sentences:

i. (a,(a,a))

Leftmost Derivation:
S ⇒ (L)
⇒ (L, S)
⇒ (S, S)
⇒ (a, S)
⇒ (a, (L))
⇒ (a, (S, S))
⇒ (a, (a, S))
⇒ (a, (a, a))

Rightmost Derivation:
S ⇒ (L)
⇒ (L, S)
⇒ (L, a)
⇒ (S, a)
⇒ (a, a)
⇒ (a, (L))
⇒ (a, (S, S))
⇒ (a, (a, a))

Parse Tree: (Tree structure depicting the derivations)

ii. (a,((a,a),(a,a)))

Leftmost Derivation:
S ⇒ (L)
⇒ (L, S)
⇒ (S, S)
⇒ (a, S)
⇒ (a, (L))
⇒ (a, (L, S))
⇒ (a, (S, S, S))
⇒ (a, ((L), S))
⇒ (a, ((S, S), S))
⇒ (a, ((a, a), S))
⇒ (a, ((a, a), (L)))
⇒ (a, ((a, a), (S, S)))
⇒ (a, ((a, a), (a, a)))

Rightmost Derivation:
(Similar steps but replacing rightmost non-terminal first)

Parse Tree: (Tree structure depicting the derivations)

3. a) Compute FIRST and FOLLOW for the grammar:
S → S S + | S S * | a

Solution:

- **FIRST(S):**
  S → S S + | S S * | a
  - From S → a, FIRST(a) = {a}
  - Since the other productions start with S, we recursively check until terminal 'a' is reached.
  - Hence, **FIRST(S) = {a}**
- **FOLLOW(S):**
  - Since S is the start symbol, $ is in FOLLOW(S).
  - S appears in the productions:
    - S S + → FOLLOW(S) includes FIRST(+) = {+}
    - S S * → FOLLOW(S) includes FIRST(*) = {*}
  - Therefore, *_FOLLOW(S) = {+, , $}_

---

3. b) Write about various types of top-down parsing. Discuss about the error recovery in predictive parsing.

Solution:

- **Types of Top-Down Parsing:**

    1. **Recursive Descent Parsing:** Uses a set of recursive procedures to process the input.
    2. **Backtracking Parsing:** Tries different alternatives until a match is found (inefficient).
    3. **Predictive Parsing:** A non-recursive method using a parsing table (efficient and preferred).
- **Error Recovery in Predictive Parsing:**
    - **Panic Mode:** Skips input symbols until a synchronizing token is found.
    - **Phrase Level:** Replaces or deletes symbols to correct errors.
    - **Error Productions:** Special grammar rules to handle common mistakes.
    - **Global Correction:** Suggests minimal changes to fix errors.

---

4. a) Give an algorithm to eliminate productions containing useless symbols and ambiguous productions from a grammar.

Solution:

- **Algorithm to Eliminate Useless Symbols:**
    1. Identify generating symbols (can derive terminal strings).
    2. Remove productions with non-generating symbols.
    3. Identify reachable symbols from the start symbol.
    4. Remove productions with unreachable symbols.
- **Ambiguity Elimination:**

    1. Modify the grammar to ensure each string has a unique parse tree.
    2. Use left factoring and eliminate left recursion to resolve ambiguity.

Question:

4b) Construct predictive parse table for the following grammar.

E → E + T | T
T → T * F | F
F → F / a | b

STACK IMPLEMENTATION AND TREE ALSO

[7M]

Answer:

Step 1: Left Recursion Elimination

Given grammar has left recursion, which needs to be eliminated for constructing a predictive parse table.

1. E → T E' E' → + T E' | ε
2. T → F T' T' → * F T' | ε
3. F → a F' | b F' → / F' | ε

Step 2: FIRST and FOLLOW Sets

- FIRST(E) = {a, b}
- FIRST(E') = {+, ε}
- FIRST(T) = {a, b}
- FIRST(T') = {*, ε}
- FIRST(F) = {a, b}
- FIRST(F') = {/, ε}
- FOLLOW(E) = {$}
- FOLLOW(E') = {$}
- FOLLOW(T) = {+, $}
- FOLLOW(T') = {+, $}
- FOLLOW(F) = {*, +, $}
- FOLLOW(F') = {*, +, $}

Step 3: Predictive Parse Table

Non-Terminal	**a**	**b**	**+**	*****	**/**	**$**
E	E → T E'	E → T E'				
E'			E' → + T E'			E' → ε
T	T → F T'	T → F T'				
T'			T' → ε	T' → * F T'		T' → ε
F	F → a F'	F → b				
F'			F' → ε	F' → ε	F' → / F'	F' → ε

Step 4: Stack Implementation Example

Parsing the string **"a + b"**:

Stack	**Input**	**Action**
E$	a + b$	Apply E → T E'
T E'$	a + b$	Apply T → F T'
F T' E'$	a + b$	Apply F → a F'
a F' T' E'$	a + b$	Match 'a'
F' T' E'$	+ b$	Apply F' → ε
T' E'$	+ b$	Apply T' → ε

Stack	Input	Action
E'$	+ b$	Apply E' → + T E'
+ T E'$	+ b$	Match '+'
T E'$	b$	Apply T → F T'
F T' E'$	b$	Apply F → b
b T' E'$	b$	Match 'b'
T' E'$	$	Apply T' → ε
E'$	$	Apply E' → ε
$	$	Accept

Step 5: Parse Tree for "a + b"

```
     E
   /   \
  T     E'
 |     /   \
 F    +     T  E'
|          |    |
a          F    ε
           |
           b
```

3. a) Define Context Free Grammar. Explain how it is suitable for parsing? Explain the recursive descent parser with example. [7M]

Context-Free Grammar (CFG): A Context-Free Grammar (CFG) is a formal set of production rules used to generate strings in a language. It consists of:

- A set of **non-terminals** (variables).
- A set of **terminals** (symbols).
- A **start symbol**.
- A set of **production rules**.

CFG is represented as G = (V, T, P, S) where:

- V = set of variables (non-terminals)
- T = set of terminals
- P = set of productions (rules)
- S = start symbol

Suitability for Parsing: CFGs are suitable for parsing because they can describe the syntax of programming languages. They help in:

- Defining nested structures (e.g., parentheses).
- Handling recursive language constructs.
- Enabling the development of parsing algorithms.

Recursive Descent Parser: It is a top-down parsing technique that consists of a set of recursive procedures to process the input. Each non-terminal has a corresponding function.

Example: Grammar: S → aSb | ε

Input: "aabb"

Parsing Steps:

1. Start with S
2. Apply S → aSb
3. Apply S → aSb again
4. Apply S → ε (empty)

Derivation: S ⇒ aSb ⇒ aaSbb ⇒ aabb

3. b) Design a non-recursive predictive parser for the following grammar: S → AaAb | BbBb
A → e
B → e
where a, b, e are terminals.

First and Follow Sets:

- First(A) = {e}
- First(B) = {e}
- First(S) = {a, b}
- Follow(S) = {$}

Parsing Table:

	a	**b**	**$**
S	AaAb	BbBb	
A	e	e	
B	e	e	

Stack Implementation (Table Format):

Stack	**Input**	**Action**
S	aabb	Apply S → AaAb
AaAb	aabb	Match 'a'

Stack	Input	Action
aAb	abb	Apply A → e
ab	abb	Match 'a'
b	bb	Match 'b'
(empty)	(empty)	Successful parsing

Parse Tree:

```
    S
  / |  \  \
 A  a   A   b
 |      |
 e      e
```

4. a) Given the following grammar:
E → E + E | E - E | E * E | E / E | -E | int
Show two different left-most derivations with the help of parse trees for the string "int + int * int / int". What does this tell you? [7M]

Left-most Derivation 1:

1. E → E + E
2. E → int + E
3. E → int + E * E
4. E → int + int * E
5. E → int + int * int / int

Parse Tree 1:

```
     E
    /|\
   E + E
  |   /|\
int  E * E
     |   |
   int   E
         |
       int
```

Left-most Derivation 2:

1. E → E + E
2. E → int + E
3. E → int + (E / E)
4. E → int + (E * E / E)
5. E → int + (int * int / int)

Parse Tree 2:

```
     E
    /|\
   E + E
  |    /|\
int  E  / E
     /|\   |
   E * E  int
  |   |
int  int
```

Observation: The different parse trees indicate that the grammar is **ambiguous**. Ambiguity means multiple parse trees (derivations) exist for the same string, which can lead to confusion in parsing.

4. b) Explain left recursion and left factoring with examples. [7M]

Left Recursion: Occurs when a non-terminal refers to itself as the leftmost symbol.

Example (Left Recursive): A → Aα | β

Issue: Causes infinite recursion in top-down parsers.

Removing Left Recursion: Rewrite as: A → βA' A' → αA' | ε

Example: E → E + T | T (Left Recursive) Rewrite as: E → T E' E' → + T E' | ε

Left Factoring: Used to transform grammar to make parsing decisions easier when multiple productions start with the same prefix.

Example (Before Factoring): A → ab | ac

After Left Factoring: A → aB B → b | c

This helps predictive parsers to avoid backtracking.

Question:

b) Define a Parser. What is the role of grammars in Parser construction? Construct the Predictive parsing table for the grammar:
G:

- **E → E + T | T**
- **T → T * F | F**
- **F → (E) | id**

Also provide:

- **STACK IMPLEMENTATION IN TABLE**

- **PARSING TREE**

Answer:

1. Definition of a Parser:

A **parser** is a component of a compiler that takes input data (usually tokens from a lexical analyzer) and constructs a syntactic structure known as a **parse tree** or **syntax tree**. It checks whether the input follows the rules of the grammar.

---

2. Role of Grammars in Parser Construction:

- A **grammar** defines the syntactic structure of a language.
- In parser construction, it helps in generating the parsing table, guiding how input tokens are processed.
- The parser uses grammar rules to build the parse tree step-by-step, verifying the correct sequence of tokens.

---

3. Left Factoring the Grammar (for Predictive Parsing):

The given grammar is **left-recursive**, which we need to eliminate for predictive parsing.

Original Grammar:

- E → E + T | T
- T → T * F | F
- F → (E) | id

Step 1: Removing Left Recursion:

- E → T E'
- E' → + T E' | ε
- T → F T'
- T' → * F T' | ε
- F → (E) | id

---

4. First and Follow Sets:

- **FIRST(E) = { (, id }**
- **FIRST(E') = { +, ε }**
- **FIRST(T) = { (, id }**
- _*FIRST(T') = { , ε }_
- **FIRST(F) = { (, id }**
- **FOLLOW(E) = {), $ }**
- **FOLLOW(E') = {), $ }**
- **FOLLOW(T) = { +,), $ }**
- **FOLLOW(T') = { +,), $ }**
- _*FOLLOW(F) = { , +,), $ }_

---

5. Predictive Parsing Table:

Non-Terminal	**(**	**id**	**+**	*****	**)**	**$**
E	E → T E'	E → T E'				
E'			E' → + T E'		E' → ε	E' → ε
T	T → F T'	T → F T'				
T'			T' → ε	T' → * F T'	T' → ε	T' → ε
F	F → (E)	F → id				

---

6. Stack Implementation in Table (Input: `id + id * id`):

Step	**Stack**	**Input**	**Action**
1	E, $	id + id * id $	Apply E → T E'
2	E', T, $	id + id * id $	Apply T → F T'
3	E', T', F, $	id + id * id $	Apply F → id
4	E', T', id, $	id + id * id $	Match id
5	E', T', $	+ id * id $	Apply T' → ε
6	E', $	+ id * id $	Apply E' → + T E'
7	E', T, +, $	+ id * id $	Match +
8	E', T, $	id * id $	Apply T → F T'
9	E', T', F, $	id * id $	Apply F → id
10	E', T', id, $	id * id $	Match id
11	E', T', $	* id $	Apply T' → * F T'
12	E', T', F, *, $	* id $	Match *
13	E', T', F, $	id $	Apply F → id
14	E', T', id, $	id $	Match id
15	E', T', $	$	Apply T' → ε

Step	Stack	Input	Action
16	E', $	$	Apply E' → ε
17	$	$	**Parsing Successful**

---

7. Parsing Tree for `id + id * id`:

```
             E
          /     \
        T       E'
       / \      /   \
     F   T'   +    T   E'
     |    |        / \    \
    id    ε      F   T'   ε
                  |    |
                 id   * F T'
                        |   |
                       id   ε
```

This covers the **definition**, **grammar role**, **predictive parsing table**, **stack implementation**, and **parsing tree** for full marks. □

a) What are the preprocessing steps required for constructing Predictive parsing table? Explain with example.

Answer:
The preprocessing steps required for constructing a predictive parsing table are:

1. **Elimination of Left Recursion:**
   Left recursion occurs when a non-terminal refers to itself as the first symbol in its own production. It should be removed to prevent infinite recursion during parsing.
2. **Left Factoring:**
   If two productions have a common prefix, they need to be factored out to help the parser make decisions based on the next input symbol.
3. **Computation of FIRST and FOLLOW Sets:**
   - **FIRST Set:** For each non-terminal, FIRST is the set of terminals that appear at the beginning of some string derived from the non-terminal.
   - **FOLLOW Set:** For each non-terminal, FOLLOW is the set of terminals that can appear immediately to the right of that non-terminal in some "sentential" form.
4. **Constructing the Parsing Table:**
   Using the FIRST and FOLLOW sets:
   - For each production A → α:
     - Add A → α to the parsing table entries corresponding to all terminals in FIRST(α).
     - If α can derive ε (empty string), add A → α to entries for all terminals in FOLLOW(A).

Example:
Consider the grammar: S → iEtS | a
E → b

- FIRST(S) = {i, a}
- FOLLOW(S) = {$} (end of input symbol)

The parsing table is constructed based on these sets.

---

a) What is an LL(1) grammar? Can you convert every context-free grammar into LL(1)? How to check if the grammar is LL(1) or not? Explain the rules.

Answer:

- **LL(1) Grammar:** A grammar is LL(1) if it can be parsed by a top-down parser that looks ahead only one symbol at a time. The first "L" stands for scanning the input from Left to right, the second "L" for producing a Leftmost derivation, and "1" for one lookahead symbol.
- **Conversion:** Not every context-free grammar can be converted into LL(1). Some grammars are inherently ambiguous or require more than one symbol of lookahead to resolve ambiguities.
- **Checking for LL(1):** A grammar is LL(1) if:
  1. For any non-terminal A, the FIRST sets of its different productions are disjoint.
  2. If ε is in FIRST(α) for some production A → α, then FIRST(α) and FOLLOW(A) must be disjoint.

Rules:

1. Eliminate left recursion.
2. Apply left factoring if needed.
3. Ensure FIRST and FOLLOW set conditions hold.

---

b) Consider the following grammar:
E → T + E | T
T → V * T | V
V → id

Write down the procedures for the non-terminals of the grammar to make a recursive descent parser.

Answer:
Recursive descent parsing involves writing a procedure for each non-terminal:

```
void E() {
    T();
    if (lookahead == '+') {
        match('+');
        E();
    }
}

void T() {
    V();
    if (lookahead == '*') {
        match('*');
        T();
    }
}

void V() {
    if (lookahead == 'id') {
        match('id');
    } else {
        error();
    }
}

void match(char expected) {
    if (lookahead == expected)
        lookahead = nextToken();
    else
        error();
}
```

Here, `lookahead` holds the current input symbol, and `match()` ensures the expected token is found.

3. a) What is an ambiguous grammar? Write a procedure to eliminate the same with an example. [7M]

Ambiguous Grammar: A context-free grammar is said to be ambiguous if there exists at least one string that can have more than one distinct parse tree or leftmost derivation.

Procedure to Eliminate Ambiguity:

1. Identify the ambiguous productions causing multiple parse trees.
2. Modify the grammar to separate expressions based on precedence and associativity rules.
3. Introduce new non-terminal symbols to handle different precedence levels.
4. Rewrite the productions to reflect these rules clearly.

Example: Ambiguous Grammar:

```
E -> E + E | E * E | (E) | id
```

For the string "id + id * id", there are two parse trees, indicating ambiguity.

Eliminating Ambiguity:

```
E -> E + T | T
T -> T * F | F
F -> (E) | id
```

Here, multiplication has higher precedence than addition, eliminating ambiguity.

---

3. b) Given the following grammar:

```
E -> E + E | E - E | E * E | E / E | - E | int
```

Show two different left-most derivations with the help of parse trees for the string "int + int * int / int". What does this infer? [7M]

Left-most Derivation 1:

```
E -> E + E
  -> int + E
  -> int + E * E
  -> int + int * E
  -> int + int * int / int
```

Parse Tree 1:

```
        E
       /|\
      E + E
      |   /|\
    int  E * E
          |   / \
        int  int / int
```

Left-most Derivation 2:

```
E -> E * E
  -> E + E * E
  -> int + E * E
  -> int + int * E
  -> int + int * int / int
```

Parse Tree 2:

```
        E
       /|\
      E * E
     /|\   |
    E + E  int
   /   \
 int   int
```

Inference: The existence of two different parse trees indicates that the grammar is ambiguous.

---

4. a) Write an algorithm for constructing Predictive parsing table. [7M]

Algorithm:

1. **Compute FIRST and FOLLOW sets:**
   - For each non-terminal, determine the FIRST set.
   - For each non-terminal, determine the FOLLOW set.
2. **Construct the table:**
   - For each production A -> α:
     - For each terminal 'a' in FIRST(α), add A -> α to M[A, a].
     - If α can derive ε (epsilon), add A -> α to M[A, b] for each 'b' in FOLLOW(A).
3. **Fill errors:**
   - Mark error entries where no productions apply.

---

4. b) Explain left recursion and left factoring with examples. [7M]

Left Recursion: A grammar is left-recursive if there exists a non-terminal A such that A → Aα | β.

Example (Left Recursion):

```
E -> E + T | T
```

Eliminating Left Recursion:

```
E -> T E'
E' -> + T E' | ε
```

Left Factoring: Left factoring is a technique to remove ambiguity when multiple productions share a common prefix.

Example (Before Left Factoring):

```
S -> if E then S else S | if E then S
```

After Left Factoring:

```
S -> if E then S S'
```

```
S' -> else S | ε
```

3. a) Consider the following grammar which is used for specifying subset of arithmetic expressions in C
A -> A - id | -A | id

i) Construct a parse tree for the string id - id - id

Parse Tree:
Since the grammar is left-recursive, we can derive the string as follows:

1. A -> A - id
2. A -> A - id - id
3. A -> id - id - id

The parse tree:

```
        A
       /|\
      A - id
     /|\
    A - id
    |
   id
```

ii) Prove that the above grammar is ambiguous

A grammar is ambiguous if there exists at least one string that can have more than one parse tree or derivation.

For the string **id - id - id**, we can have two different parse trees:

1. **Left Associative Parse Tree:**

```
        A
       /|\
      A  - id
     /|\
    A  - id
    |
   id
```

(Associates as (id - id) - id)

2. **Right Associative Parse Tree:**

```
        A
       /|\
      id  - A
           /|\
          id  - id
```

(Associates as id - (id - id))

Since the string has more than one valid parse tree, the grammar is **ambiguous**.

---

3. b) Explain with an example "why left recursion is an issue in top down parsing" and write steps for left recursion elimination

Why Left Recursion is an Issue:
Top-down parsers (like recursive descent parsers) face issues with left recursion because they may fall into infinite recursion while trying to expand the left-recursive rules.

Example:
Grammar: A -> A - id | id

- When we try to parse an input like "id - id", the parser will repeatedly apply A -> A - id without ever reaching the base case A -> id, causing an **infinite loop**.

Steps for Left Recursion Elimination:

1. Identify the left-recursive productions: A -> Aα | β
2. Replace with:
    - A -> β A'
    - A' -> α A' | ε (where ε is the empty string)

Applying to the Example:
Original: A -> A - id | id
Eliminated Left Recursion:

- A -> id A'
- A' -> - id A' | ε

This grammar is now suitable for top-down parsing.

Question:

Construct a LL(1) parsing table for the following grammar and show the working of the parser on input `-id-id-id`. Also, draw the parsing tree.

Grammar:

```
A → A - id | B
```

```
B → - A | id
```

Step 1: Left Recursion Removal

The given grammar is left-recursive. To convert it to LL(1), we must eliminate left recursion.

Original Production:

```
A → A - id | B
```

We can rewrite it as:

```
A → B A'
A' → - id A' | ε
```

The production for `B` remains the same:

```
B → - A | id
```

Step 2: FIRST and FOLLOW Sets

- **FIRST Sets:**
  - FIRST(A) = FIRST(B) = { -, `id` }
  - FIRST(A') = { -, ε }
  - FIRST(B) = { -, `id` }
- **FOLLOW Sets:**
  - FOLLOW(A) = { `$` } (assuming `$` as end marker)
  - FOLLOW(A') = { `$` }
  - FOLLOW(B) = { -, `$` }

Step 3: LL(1) Parsing Table

Non-Terminal	**`-`**	**`id`**	**`$`**
A	A → B A'	A → B A'	
A'	A' → - id A'		A' → ε
B	B → - A	B → id	

Step 4: Parsing Process for Input `- id - id - id`

Input: `- id - id - id $`
Stack: `A $`

Stack	Input	Action
A $	- id - id - id $	A → B A'

Stack	Input	Action
A' B $	- id - id - id $	B → - A
A' A - $	id - id - id $	Match –
A' A $	id - id - id $	A → B A'
A' A' B $	id - id - id $	B → id
A' A' $	id - id - id $	Match `id`
A' $	- id - id $	A' → - id A'
A' A' - $	id - id $	Match –
A' A' $	id - id $	Match `id`
A' $	- id $	A' → - id A'
A' A' - $	id $	Match –
A' A' $	id $	Match `id`
A' $	$	A' → ε
$	$	Accepted

Step 5: Parsing Tree

```
                A
              /   \
             B     A'
           /       /  \
         - A    - id  A'
         / \           |
        B  A'          ε
       /   |
     id    ε
```

This parsing tree represents the derivation of `– id – id – id` from the given grammar.

3. a) Consider the following grammar which is used for specifying logical expressions in Python:
L -> L and L | L or L | not L | TRUE | FALSE

i) Construct parse tree(s) for the string "not TRUE and FALSE"

Parse Tree 1:

```
        L
       /|\
    L  and  L
    |        |
 not L     FALSE
   |
 TRUE
```

Parse Tree 2:

```
        L
       / \
     not   L
          /|\
       L  and L
       |      |
    TRUE    FALSE
```

ii) Prove that the above grammar is ambiguous
A grammar is said to be ambiguous if there exists more than one parse tree (or derivation) for the same string. As shown above, the string "not TRUE and FALSE" has two different parse trees, hence the grammar is ambiguous.

---

3. b) Explain with an example "why common left factors is an issue in top-down parsing" and write steps for left factoring

Issue with Common Left Factors:
Common left factors create ambiguity in choosing which production to apply during top-down parsing, making it difficult to decide the correct rule based on the next input symbol.

Example:
Consider the grammar:
A -> ab | ac

While parsing an input starting with 'a', the parser cannot immediately determine whether to use 'ab' or 'ac' because both start with 'a'.

Steps for Left Factoring:

1. Identify productions with common prefixes.
2. Extract the common prefix.
3. Introduce a new non-terminal to handle the differing suffixes.

Left Factored Grammar:
A -> aB
B -> b | c

Now, the parser can easily decide to apply A -> aB upon seeing 'a', and then choose between 'b' and 'c' based on the next input symbol.

Question:

Construct a LL(1) parsing table for the following grammar and show the working of the parser on the input `not TRUE and FALSE`. Also, provide the parsing tree.

Given Grammar:

1. **L → L or B | B**
2. **B → B and C | C**
3. **C → not L | TRUE | FALSE**

Step 1: Left Factoring and Removing Left Recursion

Since the grammar has left recursion, we need to eliminate it to construct the LL(1) parsing table.

For L → L or B | B:

Left recursion removal:

- **L → B L'**
- **L' → or B L' | ε**

For B → B and C | C:

Left recursion removal:

- **B → C B'**
- **B' → and C B' | ε**

For C → not L | TRUE | FALSE (no left recursion here)

Step 2: First and Follow Sets

First Sets:

- **First(L) = {not, TRUE, FALSE}**
- **First(L') = {or, ε}**
- **First(B) = {not, TRUE, FALSE}**
- **First(B') = {and, ε}**
- **First(C) = {not, TRUE, FALSE}**

Follow Sets:

- **Follow(L) = {$, or, and, FALSE, TRUE}**

- **Follow(L') = {$, or, and, FALSE, TRUE}**
- **Follow(B) = {or, $, and}**
- **Follow(B') = {or, $, and}**
- **Follow(C) = {or, $, and}**

Step 3: LL(1) Parsing Table

Non-Terminal	not	TRUE	FALSE	or	and	$
L	L → B L'	L → B L'	L → B L'			
L'				L' → or B L'		L' → ε
B	B → C B'	B → C B'	B → C B'			
B'				B' → ε	B' → and C B'	B' → ε
C	C → not L	C → TRUE	C → FALSE			

Step 4: Parsing Input `not TRUE and FALSE`

Input: `not TRUE and FALSE $`

Stack: `L $`

Stack	Input	Action
L $	not TRUE and FALSE $	L → B L'
B L' $	not TRUE and FALSE $	B → C B'
C B' L' $	not TRUE and FALSE $	C → not L
not L B' L' $	not TRUE and FALSE $	Match `not`
L B' L' $	TRUE and FALSE $	L → B L'
B L' B' L' $	TRUE and FALSE $	B → C B'
C B' L' B' L' $	TRUE and FALSE $	C → TRUE
TRUE B' L' B' L'$	TRUE and FALSE $	Match `TRUE`
B' L' B' L' $	and FALSE $	B' → and C B'
and C B' L' B' L'$	and FALSE $	Match `and`
C B' L' B' L' $	FALSE $	C → FALSE
FALSE B' L' B' L'$	FALSE $	Match `FALSE`
B' L' B' L' $	$	B' → ε
L' B' L' $	$	L' → ε
B' L' $	$	B' → ε
L' $	$	L' → ε

Stack	Input	Action
$	$	**Accepted**

Step 5: Parsing Tree

```
                  L
                 / \
                B   L'
               / \    \
              C   B'   ε
             /     |
          not      and
           |         \
           L          C
          / \          |
         B   L'       FALSE
        / \    \
       C   B'   ε
       |
     TRUE
```

This structure shows the complete LL(1) parsing process with the parsing table and tree for the input `not TRUE and FALSE`.

3. a) Consider the following grammar which is used for specifying subset of arithmetic expressions in C, where *num* is an integer constant:

E -> num − E | num + E | num

i) **Construct a parse tree for the string:**
num − num - num + num

Parse Tree:

```
        E
      / | \
   num  -   E
          / | \
      num -   E
            / | \
         num +  E
                |
               num
```

This represents the expression parsed as:
num - (num - (num + num))

---

ii) **As per the grammar given above, what is the result of the expression:**
9 - 5 - 2 + 4

Step-by-step Evaluation (Left Associative):

- Expression: **9 - (5 - (2 + 4))**
- Step 1: **2 + 4 = 6**
- Step 2: **5 - 6 = -1**
- Step 3: **9 - (-1) = 10**

Final Result: 10

---

3. b) Explain the dangling else problem and what is the solution for it as per the C language specification.

Dangling Else Problem:
The *dangling else* problem arises in programming languages like C where nested *if-else* statements create ambiguity about which *if* statement an *else* clause should be associated with.

Example:

```
if (condition1)
    if (condition2)
        statement1;
    else
        statement2;
```

Here, it is unclear whether the `else` belongs to:

- The first `if (condition1)` **or**
- The second `if (condition2)`

Solution in C:
As per C language specification, the `else` is always associated with the **nearest unmatched `if`**. So, in the above example, `else` is paired with `if (condition2)`.

To avoid ambiguity, use braces `{}`:

```
if (condition1) {
    if (condition2) {
        statement1;
    } else {
        statement2;  // Clearly associated with condition2
    }
}
```

This ensures code clarity and eliminates the dangling else problem.

Question:

4. Construct a LL(1) parsing table for the following grammar and show the working of parser on input `aa+a*`. Also provide the parsing tree.

Given Grammar:

- S → SS+
- S → SS*
- S → a

Step 1: Left Factoring the Grammar

The given grammar is left-recursive. To convert it into LL(1), we first remove left recursion.

Original:

- S → SS+ | SS* | a

Rewrite using left factoring:

- S → aS'
- S' → S+ | S* | ε

Step 2: First and Follow Sets

1. **First(S):** { a }
2. **First(S'):** { a, +, *, ε }
3. **Follow(S):** { $, +, * } (since S is the start symbol)
4. **Follow(S'):** { $, +, * }

Step 3: Construct the LL(1) Parsing Table

Non-Terminal	**a**	**+**	*****	**$**
S	S → aS'	-	-	-
S'	S' → S+ / S*	S' → ε	S' → ε	S' → ε

- **S' → S+** when the next input is '+'
- **S' → S*** when the next input is '*'
- **S' → ε** for follow symbols if no matching terminal exists

---

Step 4: Parsing Input `aa+a*`

Input: `aa+a*`$
Stack (initial): `S`$

Step	Stack	Input	Action
1	S$	aa+a*$	S → aS'
2	S'$	a+a*$	Match 'a'
3	S'$	+a*$	S' → S+
4	S+$	+a*$	Match '+'
5	S$	a*$	S → aS'
6	S'$	*$	Match 'a'
7	S'$	*$	S' → S*
8	S*$	*$	Match '*'
9	S$	$	S → ε (input consumed)

Parsing Successful □

Step 5: Parsing Tree

```
    S
  /   \
a      S'
      /   \
     S     +
    / \
   a   S'
      /   \
     S     *
    / \
   a   S'
       |
       ε
```

This completes the LL(1) parsing table, parsing steps, and the parse tree for the given grammar.

Here's the detailed answer with questions included:

a) Consider the following grammar which is used for specifying a subset of arithmetic expressions in C, where `num` is an integer constant:

E → num * E | num / E | num

i) Construct a parse tree for the string:

num / num / num * num

Solution:
We can derive the expression using the given grammar rules.

- Start with `E`:
  - E → num / E
    - E → num / E
      - E → num * E
        - E → num

Parse Tree:

```
          E
         /|\
     num /  E
          /|\
     num /   E
          /|\
     num  *  E
             |
           num
```

This corresponds to the derivation:
`E → num / (num / (num * num))`

ii) As per the grammar given above, what is the result of the expression:

12 / 12 / 2 * 3

Solution:
The expression follows the parse tree structure derived above:
`12 / (12 / (2 * 3))`

Step-by-step evaluation:

1. **2 * 3 = 6**
2. **12 / 6 = 2**
3. **12 / 2 = 6**

Final Result: 6

---

b) Explain with an example “why ambiguity is an issue in parsing” and write an intuitive argument why it is difficult to solve.

Solution:

Ambiguity in Parsing:
A grammar is **ambiguous** if there exists at least one string that can have **more than one parse tree** (or derivation). This causes confusion in understanding the structure and meaning of the expression.

Example:
Consider the simple grammar for arithmetic expressions:

- E → E + E | E * E | num

For the expression **2 + 3 * 4**, we can derive two parse trees:

1. **(2 + 3) * 4**
2. **2 + (3 * 4)**

Both are valid according to the grammar, but they yield different results:

- (2 + 3) * 4 = 5 * 4 = 20
- 2 + (3 * 4) = 2 + 12 = 14

This ambiguity leads to **different outcomes** based on how the expression is parsed.

Why Ambiguity is Difficult to Solve:

1. **Multiple Interpretations:** Some languages naturally allow multiple ways to interpret the same expression, making it hard to define strict, unambiguous rules.
2. **Complex Grammar Structures:** As grammars grow complex (with nested operations, precedence, associativity), detecting and resolving ambiguities becomes computationally challenging.
3. **Undecidability:** In general, it’s **undecidable** to determine whether any arbitrary context-free grammar is ambiguous. There's no universal algorithm to solve this for all grammars.

Hence, ambiguity is a critical issue because it affects **compiler design**, **expression evaluation**, and **program correctness**.

Question:

Construct a LL(1) parsing table for the following grammar and show the working of the parser on input `((a,a),a,(a))`. Also, draw the parsing tree.
Grammar:

- S → (L) | a
- L → L, S | S

Step 1: Compute FIRST and FOLLOW sets

FIRST Sets:

- FIRST(S) = { '(', 'a' }
- FIRST(L) = { '(', 'a' }

FOLLOW Sets:

- FOLLOW(S) = { '$', ',', ')' }
- FOLLOW(L) = { ')' }

Step 2: Eliminate Left Recursion (for L → L, S | S)

Convert to:

- L → S L'
- L' → , S L' | ε

Now the grammar is:

- S → (L) | a
- L → S L'
- L' → , S L' | ε

Step 3: FIRST and FOLLOW Sets After Modification

- FIRST(S) = { '(', 'a' }
- FIRST(L) = { '(', 'a' }
- FIRST(L') = { ',', ε }
- FOLLOW(S) = { '$', ',', ')' }
- FOLLOW(L) = { ')' }
- FOLLOW(L') = { ')' }

Step 4: LL(1) Parsing Table

Non-Terminal	(	a	,	)	$
S	S → (L)	S → a			
L	L → S L'	L → S L'			
L'			L' → , S L'	L' → ε	

Step 5: Parsing the Input `((a,a),a,(a))`

Input: `((a,a),a,(a))$`
Stack: Start with `S$`

Stack	Input	Action
S$	((a,a),a,(a))$	Apply S → (L)
(L)$	((a,a),a,(a))$	Match '('
L)$	(a,a),a,(a))$	Apply L → S L'
S L')$	(a,a),a,(a))$	Apply S → (L)
(L) L')$	(a,a),a,(a))$	Match '('
L) L')$	a,a),a,(a))$	Apply L → S L'
S L') L')$	a,a),a,(a))$	Apply S → a
a L') L')$	a,a),a,(a))$	Match 'a'
L') L')$	,a),a,(a))$	Apply L' → , S L'
, S L' L')$	,a),a,(a))$	Match ','
S L' L')$	a),a,(a))$	Apply S → a
a L' L')$	a),a,(a))$	Match 'a'
L' L')$	),a,(a))$	Apply L' → ε
L')$	),a,(a))$	Match ')'
L')$	,a,(a))$	Apply L' → , S L'
, S L'$	,a,(a))$	Match ','
S L'$	a,(a))$	Apply S → a
a L'$	a,(a))$	Match 'a'
L'$	,(a))$	Apply L' → , S L'
, S L'$	,(a))$	Match ','
S L'$	(a))$	Apply S → (L)
(L) L'$	(a))$	Match '('
L) L'$	a))$	Apply L → S L'
S L') L'$	a))$	Apply S → a

Stack	Input	Action
a L') L'$	a))$	Match 'a'
L') L'$	))$	Apply L' → ε
L'$	))$	Match ')'
$	$	Successfully Parsed!

Step 6: Parsing Tree

```
        S
       / \
      (   )
       |
       L
     /   \
    S    L'
   / \     \
  (   )     ,
   |        \
   L         S
 /   \        \
S    L'        a
|      \
a       ,
       / \
      S  L'
      |   |
      a   ε
```

□ Final Answer Includes:

1. **Question**
2. FIRST and FOLLOW Sets
3. Left Recursion Removal
4. LL(1) Parsing Table
5. Parsing Process (Step-by-step)
6. Parsing Tree