Q1. Define Compiler, Translator, and Interpreter. Give one example of each.

🔹 **1. Compiler – Definition, Working, and Example**

Definition:

A compiler is a program that translates the entire source code of a high-level programming language (like C, C++, or Java) into machine code (object code or binary code) before the program is executed.

The compiler performs several phases of translation such as lexical analysis, syntax analysis, semantic analysis, optimization, and code generation, producing an executable file.

Working of a Compiler:

When you compile a program, the compiler performs these main stages:

1. **Lexical Analysis:**
   Breaks the source code into tokens (keywords, identifiers, operators, literals).
   Example:

   int sum = a + b;

   Tokens: int, sum, =, a, +, b, ;
2. **Syntax Analysis:**
   Checks the structure using grammar rules. Detects syntax errors.
3. **Semantic Analysis:**
   Ensures that variables and operations make logical sense (e.g., type checking).
4. **Intermediate Code Generation:**
   Converts to a machine-independent code (like three-address code).
5. **Code Optimization:**
   Improves efficiency by removing redundant code.
6. **Code Generation:**
   Produces final machine-level or assembly-level code.
7. **Code Linking & Loading:**
   Links external libraries and prepares executable output.

Example:

- **Compiler: GCC (GNU C Compiler)**
  - **Used for compiling C and C++ programs.**
  - **Command: gcc program.c -o program**

Input: program.c (C source file)
Output: program.exe (Executable machine code file)

Advantages of a Compiler:

- **Translates the entire program only once → faster execution afterward.**
- **Performs error detection during compilation → helps debugging early.**
- **Generates optimized executable code.**

Disadvantages:

- **Requires time for initial compilation.**
- **Must recompile after every change in source code.**

🔹 **2. Translator – Definition, Working, and Example**

Definition:

A translator is a general term for any program that converts code written in one language into another language — usually from a high-level programming language to a lower-level language (like assembly or machine code).

Hence, compiler, assembler, and interpreter are all types of translators.

Types of Translators:

1. **Assembler: Converts assembly language to machine code.**
2. **Compiler: Converts high-level code to machine code (whole program).**
3. **Interpreter: Converts high-level code to machine code (line by line).**

Advantages of a Translator:

- **Enables portability between languages.**
- **Reduces manual conversion effort.**
- **Simplifies program execution on different machines.**

Disadvantages:

- **Adds one extra step before execution.**
- **May not always optimize code effectively.**

🔹 **3. Interpreter – Definition, Working, and Example**

Definition:

An interpreter is a program that translates and executes source code line by line.
It does not generate a separate machine code file; instead, it directly runs each instruction as it is read.

This means that if the program has an error on line 10, the interpreter executes lines 1–9 and stops when it reaches line 10.

Working of an Interpreter:

1. **Reads one line (or instruction) from the source code.**
2. **Checks its syntax and semantics.**
3. **Converts it to machine code temporarily and executes it immediately.**
4. **Moves to the next line.**

Flow:

Source Code → Interpreter → Output

Example:

- **Interpreter: Python Interpreter (CPython)**
    - **Command: python program.py**

Input:

```
x = 10
y = 5
print(x + y)
```

Output:

```
15
```

No separate machine code file is produced — the interpreter executes it directly.

Advantages of an Interpreter:

- **Easier to debug since it stops immediately at an error.**
- **Allows immediate execution (used in scripting languages).**
- **Saves compilation time for small programs.**

Disadvantages:

- **Slower execution because translation happens every time the program runs.**
- **No optimized machine code is generated.**

---

2. What is Bootstrapping?

Definition:

Bootstrapping is the process of building a compiler using the same programming language for which the compiler is designed.
In simple terms, it means a compiler that compiles itself.

For example:
If we write a C compiler in C language, the process of creating that compiler is called bootstrapping.

Why Bootstrapping is Needed:

Bootstrapping is essential because when a new programming language is developed, we must also develop its compiler.
But initially, no compiler exists for that new language, so we have two options:

1. **Write the first compiler in another existing language (like Assembly, C, etc.), or**
2. **Use bootstrapping to develop the compiler using the same language itself.**

Bootstrapping allows the language to evolve independently, and developers can modify or improve it without depending on another language.

Basic Idea Behind Bootstrapping:

Let's say we want to create a compiler for a language called L.

1. **First, we write a simple version of the compiler for language L in another known language, say Assembly or C.**
2. **This small compiler can compile a better version of the compiler written in L itself.**
3. **Once the improved compiler is compiled and executed successfully, it can be used to compile its own source code and other programs written in L.**

This process is self-sustaining, and we can keep improving the compiler step by step.

Steps in Bootstrapping:

The bootstrapping process usually occurs in three main stages:

Stage 1: Initial Compiler (Machine Dependent)

- **A small and simple compiler (called a cross-compiler) is written in another existing language (like Assembly or C).**
- **This compiler can translate programs written in the new language L into machine code.**

Stage 2: Improved Compiler (Written in Target Language L)

- **Using the initial compiler, we compile an improved compiler written in L itself.**
- **This version has more features and is more efficient.**

Stage 3: Self-Compilation

- **Now, the improved compiler is capable of compiling its own source code.**
- **This means the compiler is self-hosting, and the language has become independent of the original language.**

Example of Bootstrapping:

Let's take a real-world example:

1. **Suppose we are developing a C compiler.**
2. **We first write a basic compiler for C in Assembly language.**
3. **Then, we write an improved version of the compiler in C itself.**
4. **Using the first compiler (written in Assembly), we compile this new compiler.**
5. **Once it works, we can use this new compiler to compile its own source code and future versions.**

That's bootstrapping in action!

Advantages of Bootstrapping:

1. ✅ **Language Independence:**
    - **Once bootstrapped, the compiler no longer depends on another language.**
2. ✅ **Easier Maintenance and Upgradation:**
    - **Since the compiler is written in its own language, developers can modify and extend it easily.**
3. ✅ **Portability:**
    - **Bootstrapped compilers can be easily moved to different platforms or architectures.**
4. ✅ **Proof of Language Power:**
    - **If a compiler can compile itself, it proves that the language is powerful enough to implement complex software.**

Disadvantages of Bootstrapping:

1. ❌ **Initial Effort is High:**
    - **Writing the first minimal compiler in another language can be time-consuming.**
2. ❌ **Complex Debugging:**
    - **Errors in the self-compiled compiler can be hard to detect and fix.**

3. ❌ **Resource Intensive:**
    - **It may require significant computational resources to compile multiple versions.**

Real-Life Examples:

1. **C Compiler:**
    - **The original C compiler was written in Assembly, and later bootstrapped to be written in C itself.**
2. **Java Compiler (javac):**
    - **The Java compiler (javac) is written in Java — another example of bootstrapping.**
3. **Rust, Go, Swift, and Pascal:**
    - **These modern languages also use bootstrapping for building self-hosting compilers.**

---

3. Define Token and Explain the Role of the Lexical Analyzer

🧩 **Definition of Token**

A token is the smallest meaningful unit in a program that has a collective meaning to the compiler. It represents a category or type of lexical unit in the source program.

In other words:

A token is a sequence of characters that represents a basic syntactic unit such as a keyword, identifier, operator, constant, or symbol.

Each token has two parts:

1. **Token name (type) – represents the category of token (e.g., identifier, keyword, operator).**
2. **Attribute value (lexeme) – the actual text (value) from the source code.**

Example:

Consider this simple C statement:

sum = a + b;

Lexeme	Token Name	Description
sum	Identifier	Variable name
=	Assignment Operator	Assigns value

Lexeme	Token Name	Description
a	Identifier	Variable name
+	Operator	Addition
b	Identifier	Variable name
;	Delimiter	Statement terminator

Here, the lexical analyzer divides the code into these tokens and sends them to the syntax analyzer for further processing.

Types of Tokens:

1. **Keywords:**
   Reserved words that have predefined meaning.
   Example: if, else, for, while, return.
2. **Identifiers:**
   Names given to variables, functions, arrays, etc.
   Example: sum, count, main.
3. **Constants / Literals:**
   Fixed values that do not change.
   Example: 10, 3.14, 'A', "Hello".
4. **Operators:**
   Symbols used to perform operations.
   Example: +, -, *, /, =, ==.
5. **Punctuations / Delimiters:**
   Used for separating program elements.
   Example: ;, ,, {, }, (,).
6. **Strings:**
   Sequence of characters enclosed in quotes.
   Example: "Hello World".

⚙️ **Lexical Analyzer: Definition and Role**

Definition:

A lexical analyzer, also known as a scanner, is the first phase of a compiler.
It reads the source code character by character, groups them into tokens, and passes these tokens to the syntax analyzer (parser).

It also removes unnecessary information like white spaces, comments, and tabs.

Diagram:

Source Code

↓

Lexical Analyzer

↓

Tokens → Syntax Analyzer

Main Roles / Functions of Lexical Analyzer:

1. Tokenization

- **The primary task of the lexical analyzer is to scan the source code and divide it into tokens.**
- **It identifies sequences like keywords, identifiers, numbers, and operators.**

2. Removing White Spaces and Comments

- **It ignores blank spaces, tabs, and comments since they are not needed for syntax analysis.**
- **Example: Comments like /* This is a comment */ are discarded.**

3. Interaction with Symbol Table

- **When an identifier or constant is found, the lexical analyzer enters it into the symbol table.**
- **The symbol table stores variable names, data types, and addresses.**

4. Error Detection (Lexical Errors)

- **Detects lexical errors like invalid characters or malformed numbers.**
- **Example: 9abc is invalid as an identifier.**

5. Communication with Syntax Analyzer

- **It passes tokens to the parser for syntactic analysis.**
- **If the syntax analyzer requests the next token, the lexical analyzer provides it.**

6. Buffering and Input Management

- **It uses buffers to efficiently read large inputs from the source file.**
- **Helps in improving compiler performance.**

Example of Lexical Analysis Process:

Let's take a C statement:

```
int total = a + 5;
```

Step-by-step breakdown:

Step	Lexeme	Token
1	int	Keyword
2	total	Identifier
3	=	Operator
4	a	Identifier
5	+	Operator
6	5	Constant
7	;	Delimiter

All unnecessary spaces and comments are ignored.

Errors Detected by Lexical Analyzer:

1. **Invalid token: Example – @name (invalid character @)**
2. **Unterminated string: Example – "Hello (missing closing quote)**
3. **Illegal number format: Example – 12.3.4**
4. **Unknown characters: Example – $ if not supported by the language**

Advantages of Lexical Analysis:

1. ✅ **Simplifies the parsing process by providing a stream of tokens.**
2. ✅ **Improves performance through buffering and lookahead.**
3. ✅ **Enhances readability by removing unnecessary details.**
4. ✅ **Detects lexical errors early in the compilation process.**

Difference Between Token, Lexeme, and Pattern:

Term	Definition	Example
Token	Category of lexical unit	Identifier
Lexeme	Actual text in the source code	sum, a, b
Pattern	Rule or regular expression defining a token	[A-Za-z][A-Za-z0-9]* for identifiers

---

4. What is Ambiguity in Grammar? Give One Example of an Ambiguous Grammar.

🧠 **Introduction**

In compiler design, grammars play a vital role in defining the syntactic structure of programming languages.
A grammar describes how valid statements of a language can be formed using rules of production.

However, sometimes a grammar can generate more than one parse tree (or derivation tree) for the same input string.
Such a grammar is said to be ambiguous.

🔍 **Definition of Ambiguity**

A grammar is said to be ambiguous if there exists at least one string that can be generated by the grammar in more than one way, i.e., it can have two or more distinct parse trees or derivations.

In simple terms:

A grammar is ambiguous if a single statement or expression can be interpreted in more than one way by the compiler.

Formally:

A context-free grammar (CFG) G = (V, T, P, S) is said to be ambiguous if there exists a string w ∈ L(G) such that:

- **w has two or more different leftmost derivations, or**
- **w has two or more different parse trees.**

⚙️ **Reason Behind Ambiguity**

Ambiguity mainly occurs due to:

1. **Lack of clear precedence or associativity between operators.**
2. **Multiple production rules that can generate the same string.**
3. **Improper grammar structure that does not define clear grouping of expressions.**

🧩 **Example of Ambiguous Grammar**

Consider the grammar G defined as:

E → E + E | E * E | (E) | id

This grammar can generate arithmetic expressions involving +, *, and parentheses.
Let's check the ambiguity for a specific input string.

Example String:

id + id * id

Now, using the grammar above, this string can be parsed in two different ways depending on how we group the operators.

Derivation 1 (Interpret as (id + id) * id)

1. **E → E * E**
2. **E → E + E**
3. **E → id**
4. **Combine to get: (id + id) * id**

Parse Tree 1:

mathematica Copy code

```
        E
      /|\
     E * E
    /|\
   E + E
  |   |
 id   id
        \
         id
```

Derivation 2 (Interpret as id + (id * id))

1. E → E + E
2. First E → id
3. Second E → E * E
4. Combine to get: `id + (id * id)`

Parse Tree 2:

mathematica Copy code

```
      E
     /|\
    E + E
   |   /|\
  id  E * E
      |   |
     id   id
```

✅ **Both derivations generate the same string:**
id + id * id
But they produce different parse trees, representing different meanings.

- **Parse Tree 1 → id + id evaluated first, then multiplied by id.**
- **Parse Tree 2 → id * id evaluated first, then added to id.**

Thus, the grammar is ambiguous.

🚦 **Why is Ambiguity a Problem?**

1. **Multiple Interpretations:**
   The compiler cannot decide which meaning to choose.
   For example, in arithmetic expressions, ambiguity affects operator precedence and associativity.
2. **Incorrect Code Translation:**
   The compiler may generate incorrect intermediate or machine code.
3. **Error-Prone Parsing:**
   Ambiguous grammars complicate the parsing process, making syntax analysis unreliable.
4. **Optimization Issues:**
   Ambiguity can affect how expressions are optimized during compilation.

🧰 **Removing Ambiguity**

Ambiguity can be removed by:

1. **Rewriting the grammar with clear operator precedence and associativity.**
2. **Using separate non-terminals for expressions with different precedence levels.**
3. **Using unambiguous parsing methods (like LL(1), LR(1)).**

Example of Removing Ambiguity

Let's remove ambiguity from the previous grammar.

Ambiguous grammar:

E → E + E | E * E | (E) | id

Unambiguous grammar (defines precedence of * over +):

E → E + T | T

T → T * F | F

F → (E) | id

Now:

- *** has higher precedence than +.**
- **Multiplication is performed before addition.**
- **Only one parse tree will be generated for each valid string.**

For example, for id + id * id,
this grammar ensures id * id is evaluated before addition.

---

5. Explain the Phases of a Compiler in Detail with a Diagram

(14 Marks Answer – Fully Detailed)

🧠 Introduction

A compiler is a special program that translates a high-level language (HLL) program (like C, C++, or Java) into machine-level code that can be understood and executed by a computer.

The process of compilation is not done in one single step — instead, it is divided into multiple well-defined phases, each having a specific function.

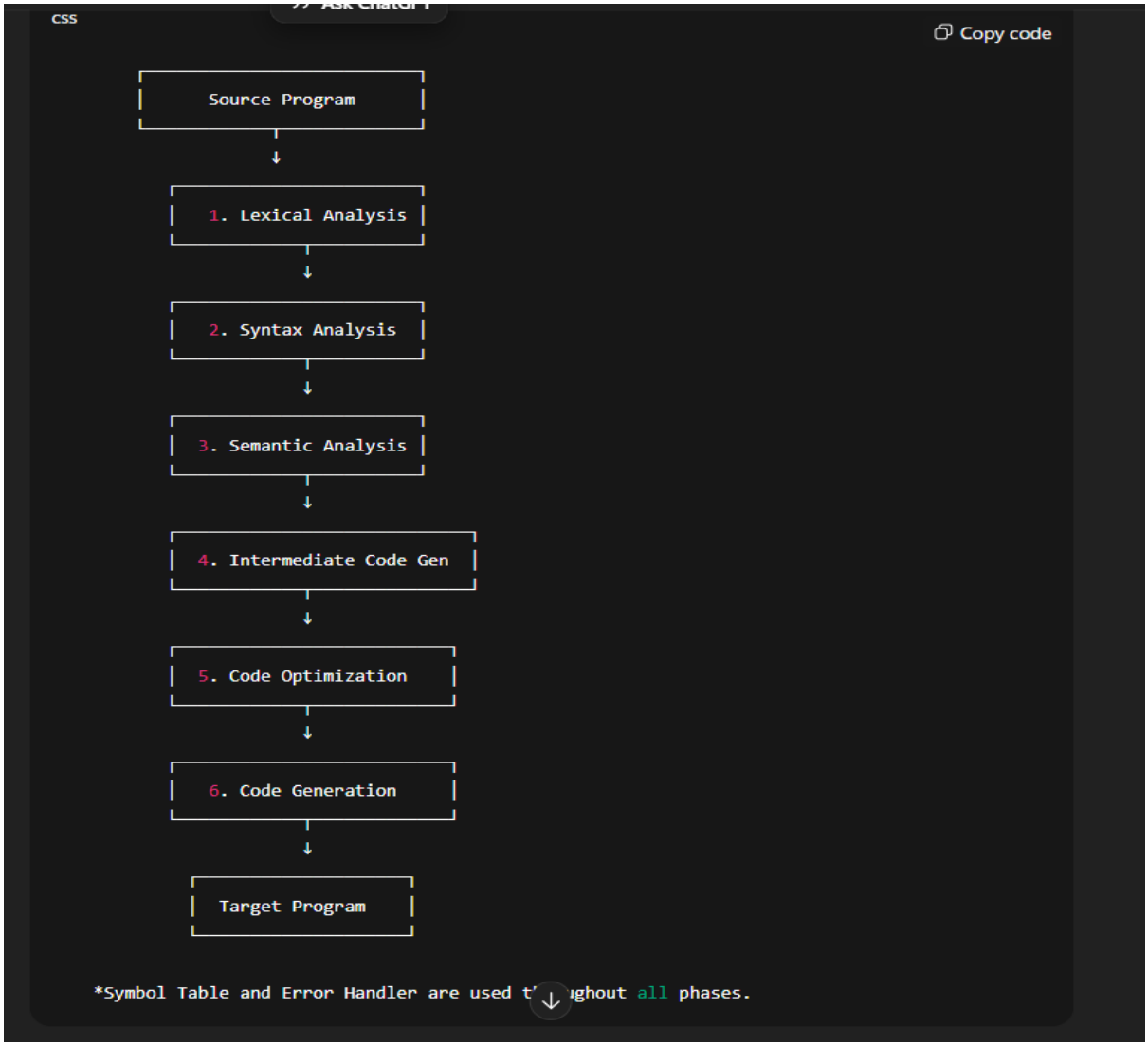
These phases work sequentially, and the output of one phase becomes the input to the next phase. Together, they form the compiler structure or compiler architecture.

⚙️ Phases of a Compiler

The compiler mainly consists of six major phases, which can be grouped into two main parts:

1. **Analysis Phase (Front End)**
    - **Understands and analyzes the source program.**
    - **Divides it into tokens, checks syntax and semantics, and produces an intermediate representation (IR).**
2. **Synthesis Phase (Back End)**
    - **Takes the intermediate representation and generates optimized machine code.**

📘 Diagram: Phases of a Compiler



🧩 **Detailed Explanation of Each Phase**

1️⃣ **Lexical Analysis (Scanner Phase)**

- **Purpose: Converts the raw source code into a sequence of tokens.**
- **Input: Source program (characters).**
- **Output: Tokens.**
- **Main Task: Remove whitespace, comments, and group characters into tokens such as keywords, identifiers, operators, constants, etc.**

Example:

int a = b + c;

Tokens generated → int, a, =, b, +, c, ;

Responsibilities:

- **Scanning the input text.**

- **Building a symbol table (for identifiers).**
- **Handling lexical errors (like invalid symbols or identifiers).**

Tool used: Lex or Flex (for automatic lexical analyzer generation).

2️⃣ Syntax Analysis (Parser Phase)

- **Purpose: Checks the grammatical structure of the source code using context-free grammar (CFG).**
- **Input: Stream of tokens.**
- **Output: Parse tree or syntax tree.**

Main Role:
Ensures that the sequence of tokens follows the language grammar.

Example:
For the expression a + b * c,
the parser checks operator precedence and forms a syntax tree showing multiplication before addition.

Errors Detected:

- **Missing semicolons**
- **Mismatched parentheses**
- **Invalid expressions**

Tool used: YACC, Bison (parser generators).

3️⃣ Semantic Analysis

- **Purpose: Ensures semantic correctness of the program.**
- **Input: Parse tree.**
- **Output: Annotated syntax tree (with meaning attached).**

Main Task:
Checks whether the statements make logical sense.

Checks Performed:

- **Type checking: e.g., int a = "hello"; is invalid.**
- **Scope resolution: Ensures variables are declared before use.**
- **Function checking: Validates function calls and return types.**

Example:

```
int a;
```

a = "abc"; // Semantic Error (string assigned to int)

4️⃣ Intermediate Code Generation

- **Purpose: Converts the syntax tree into a machine-independent intermediate representation (IR).**
- **Input: Annotated syntax tree.**
- **Output: Intermediate code (like three-address code, quadruples, or syntax-directed translation).**

Example:
For a + b * c, intermediate code may be:

```
t1 = b * c
t2 = a + t1
```

This code is not machine-specific but is easy to optimize and translate into machine instructions later.

5️⃣ Code Optimization

- **Purpose: Improves the performance and efficiency of the code without changing its meaning.**
- **Input: Intermediate code.**
- **Output: Optimized intermediate code.**

Types of Optimization:

1. **Local optimization: Within a single basic block.**
2. **Global optimization: Across multiple blocks or loops.**

Examples of Optimization:

- **Constant folding: x = 2 + 3 → x = 5**
- **Dead code elimination: Removing code that never executes.**
- **Loop optimization: Reducing redundant calculations inside loops.**

Optimization is crucial for making programs run faster and consume less memory.

6️⃣ Code Generation

- **Purpose: Converts optimized intermediate code into machine code or assembly code.**
- **Input: Optimized intermediate code.**
- **Output: Target (machine) code.**

Tasks Performed:

- **Register allocation and instruction selection.**
- **Memory management and addressing.**
- **Generating binary or executable files.**

Example:

t1 = b * c → MUL R1, b, c

t2 = a + t1 → ADD R2, a, R1

Supporting Components

(a) Symbol Table

- **A data structure used throughout all phases.**
- **Stores information about identifiers such as variable names, data types, scope, and memory addresses.**

Example:

Name	Type	Scope	Address
a	int	local	1000
b	int	local	1004

(b) Error Handler

- **Detects and reports errors in each phase:**
  - **Lexical errors: Invalid token.**
  - **Syntax errors: Missing operators, wrong punctuation.**
  - **Semantic errors: Type mismatch, undeclared variables.**
  - **Runtime errors: Division by zero.**
- **Helps the compiler continue analyzing after error detection (known as error recovery).**

1. LL(1) Parser

Introduction:

An LL(1) Parser is a type of Top-Down Parser used for syntax analysis in compilers.
It reads the input from Left to Right (first L) and constructs a Leftmost derivation (second L) of the

sentence using 1 lookahead symbol (1).
It is table-driven and non-recursive, meaning it uses a parsing table instead of function calls.

Characteristics:

- **It works on context-free grammars (CFGs).**
- **The grammar must be non-left-recursive and left-factored.**
- **At each step, it uses one symbol of lookahead to decide which production rule to apply.**
- **It uses a stack and parsing table (M).**

Algorithm / Working Steps:

1. **Initialize Stack:**
   Push $ (end marker) and start symbol S onto the stack.
   Stack → $ S
2. **Input Buffer:**
   Contains input string followed by $.
   Example: id + id $
3. **Parsing Table Construction:**
   Construct table M[A, a], where:
   - **A = Non-terminal**
   - **a = Terminal (lookahead symbol)**

Steps for filling:

- **For each production A → α,**
  add α to M[A, a] where a ∈ FIRST(α).
- **If ε ∈ FIRST(α),**
  then for each b ∈ FOLLOW(A), add A → α to M[A, b].

4. **Parsing Process:**
   - **Repeat until stack top = $:**
     - **Let X be top of stack and a be current input symbol.**
     - **If X == a, pop stack and advance input.**
     - **Else if X is a non-terminal:**
       - **Look at M[X, a].**
       - **If entry exists → Replace X with RHS of production.**
       - **Else → Report syntax error.**
     - **Else → Report error.**

Example:

Grammar:

```
r

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

Input: `id + id * id $`

Stack actions (simplified):

```
bash

Stack        Input        Action
$E           id+id*id$    Expand E → TE'
$E'T         id+id*id$    Expand T → FT'
...
```

This continues until the stack and input both contain `$`, meaning parsing is successful.

Advantages:

- **Simple and deterministic.**
- **Error detection is easy.**
- **No need for backtracking.**

Limitations:

- **Grammar must be LL(1) (non-left-recursive, left-factored).**
- **Cannot handle ambiguous grammars.**

Diagram:

2. Recursive Descent Parsing

Introduction:

A Recursive Descent Parser is a top-down parser that consists of a set of recursive procedures or functions, one for each non-terminal in the grammar.
It directly uses recursion to perform derivations and parsing.

Working Principle:

Each non-terminal is represented by a function.
When a rule is called, the function attempts to match the input tokens according to the production rules.
If the match fails, backtracking may occur (in general recursive descent).

However, for predictive recursive descent parsing, backtracking is not needed — it's used with LL(1) grammars.

Algorithm:

1. **Start with the start symbol function (e.g., E()).**
2. **Each function corresponds to a grammar rule.**
3. **Inside the function:**
    - **Check the next input token.**
    - **Decide which production to use (using lookahead).**
    - **Call other functions for non-terminals.**
    - **Match and consume tokens for terminals.**
4. **If parsing reaches the end successfully, input is accepted.**

Advantages:

- **Easy to implement manually.**
- **Matches grammar structure directly.**
- **Good for small compilers or interpreters.**

Disadvantages:

- **Needs left-recursion removal.**
- **May require backtracking if grammar is not LL(1).**
- **Large grammars become complex to handle.**

Example:

Grammar:

```
r

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

Copy code

Functions:

```cpp
void E() {
   T();
   Eprime();
}

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

void T() {
   F();
   Tprime();
}

void F() {
   if (lookahead == '(') {
      match('(');
      E();
      match(')');
   } else if (lookahead == 'id') {
      match('id');
   }
}
```

Copy code

Q7. Explain the Construction of SLR, Canonical LR, and LALR Parsing Tables

---

1. Introduction

In bottom-up parsing, the parser tries to reduce the input string to the start symbol by applying grammar productions in reverse.
To perform this efficiently, we use LR parsers, which are the most powerful class of deterministic parsers.

LR stands for:

- **L → Input is read from Left to Right**
- **R → Produces Rightmost derivation in reverse**

There are three main types of LR parsers:

1. **SLR Parser (Simple LR)**
2. **Canonical LR Parser**
3. **LALR Parser (Look-Ahead LR)**

Each one differs in how they construct and use parsing tables.

---

2. Components of LR Parsing Table

All LR parsers have two tables:

- **ACTION Table → Specifies shift, reduce, accept, or error actions.**
- **GOTO Table → Used when a non-terminal is on top of the stack (state transitions).**

Input	Action
shift s	Push next state and symbol
reduce A → α	Replace α by A
goto	Move to a new state after reduction
accept	Parsing successful

---

3. Construction Steps (General for All LR Parsers)

1. **Augment the Grammar:**
   Add a new start production:
   S' → S
2. **Compute LR(0) Items:**
   For each production A → αβ, create items like:
   - **[A → .αβ] → before parsing α**

- **[A → α.β] → after parsing α**

3. **Compute Closure:**
   If an item [A → α . B β] has a non-terminal B immediately after .,
   then include all productions of B (with . at the beginning).
4. **Compute GOTO:**
   For each item set I and symbol X,
   GOTO(I, X) gives the next item set after moving . past X.
5. **Build the DFA of Item Sets:**
   Each state in DFA corresponds to a set of LR items.
6. **Fill Parsing Table (ACTION & GOTO):**
   - **If item [A → α . a β] and a is a terminal → ACTION[I, a] = shift j**
   - **If item [A → α .] → ACTION[I, a] = reduce A → α (for all a in FOLLOW(A))**
   - **If item [S' → S .] → ACTION[I, $] = accept**
   - **For each GOTO(I, A) = J → GOTO[I, A] = J**

---

4. SLR(1) Parser (Simple LR)

Definition:

SLR is the simplest type of LR parser.
It uses FOLLOW sets to decide reduce actions.

Construction Steps:

1. **Augment the grammar:**
   S' → S
2. **Compute LR(0) items and DFA of item sets.**
3. **Construct Parsing Table:**
   - **Use FOLLOW(A) to decide where to reduce by A → α.**
4. **Resolve Conflicts:**
   - **If two actions are possible for the same cell, the grammar is not SLR(1).**

Example:

Grammar:

S → C C

C → c C | d

Augmented Grammar:

S' → S

S → C C

C → c C

C → d

Then we compute LR(0) items, closures, and goto transitions,
and use FOLLOW(C) = {c, d, $} to fill the ACTION table.

Advantages:

- **Simple and easy to construct manually.**
- **Works well for small grammars.**

Disadvantages:

- **Uses only FOLLOW sets → can cause conflicts (not powerful enough).**
- **Fails for more complex grammars.**

---

5. Canonical LR(1) Parser

Definition:

The Canonical LR parser is the most powerful LR parser.
It uses LR(1) items, which include lookahead symbols.

Each item has the form:

[A → α . β, a]

where a is a lookahead symbol that may follow A.

Construction Steps:

1. **Augment the grammar.**
2. **Construct LR(1) items:**
   Each item contains a lookahead terminal.
3. **Compute Closure(I):**
   If [A → α . B β, a] is in I,
   then for each production B → γ, and for each b ∈ FIRST(βa),
   include [B → . γ, b].
4. **Compute GOTO(I, X):**
   Move the . past symbol X.
5. **Build DFA of LR(1) items (each with lookaheads).**
6. **Build ACTION & GOTO tables:**
   - **Shift and reduce entries use specific lookaheads.**

Advantages:

- **Most powerful (handles almost all deterministic grammars).**
- **Detects errors at the earliest point.**

Disadvantages:

- **Construction is large and complex (many item sets).**
- **Not practical for large grammars (hundreds of states).**

---

6. LALR(1) Parser (Look-Ahead LR)

Definition:

The LALR parser is a simplified version of the Canonical LR parser.
It combines states in the LR(1) parser that have the same LR(0) core.

Thus, it keeps the power of LR(1) but uses fewer states.

Construction Steps:

1. **Start from the Canonical LR(1) collection.**
2. **Find item sets with identical LR(0) parts.**
3. **Merge them into a single set.**
    - **Union their lookaheads.**
4. **Construct new ACTION & GOTO tables.**

Example:

If Canonical LR(1) has these two item sets:

I1: [A → α ., a]

I2: [A → α ., b]

LALR merges them into:

I: [A → α ., {a, b}]

Advantages:

- **Compact table (like SLR).**
- **Power (almost like Canonical LR).**
- **Used in practical parser generators like YACC and Bison.**

Disadvantages:

- **May still fail for some ambiguous grammars.**
- **Slightly more complex to construct than SLR.**

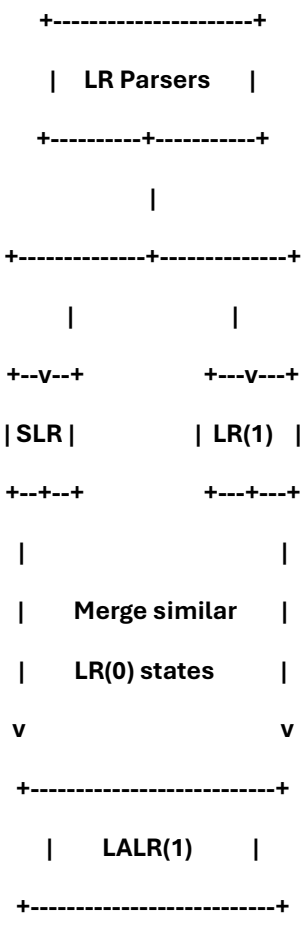
---

7. Comparison Table

Feature	SLR(1)	Canonical LR(1)	LALR(1)
Item type	LR(0)	LR(1)	LR(1) (merged)
Lookahead used	FOLLOW set	Individual lookahead	Combined lookahead
Number of states	Small	Very large	Moderate
Power	Weak	Strongest	Nearly as strong
Used in practice	Rare	Rare	Common (e.g. YACC)

---

8. Diagrammatic Summary (Text Representation)



Q8. Write short notes on the following:

(a) Three-Address Code (TAC) for Control Structures

(b) Directed Acyclic Graph (DAG)

(a) Three-Address Code (TAC) for Control Structures

Introduction:

In the Intermediate Code Generation phase, compilers convert high-level language constructs into a simpler, machine-independent form called Three-Address Code (TAC).
Each instruction in TAC involves at most three operands (hence the name *three-address*).
It helps in optimization and target code generation.

General Form of TAC:

vbnet Copy code

```
x = y op z
x = op y
x = y
goto label
if x relop y goto label
param x
call p, n
return x
```

Where:

- `x, y, z` → variables or temporaries
- `op` → arithmetic operator (`+, -, *, /`)
- `relop` → relational operator (`<, >, ==, !=` , etc.)
- `label` → a point in code to jump to

1. Control Structures in TAC

Control structures include selection statements (if, if-else) and looping statements (while, for, do-while).

(i) If-Else Statement

High-level code:

c

Copy code

```c
if (a < b)
    x = y + z;
else
    x = y - z;
```

TAC Representation:

vbnet

Copy code

```vbnet
if a < b goto L1
goto L2
L1: x = y + z
    goto L3
L2: x = y - z
L3:
```

Explanation:

- Conditional jump is used to go to the "true" block.
- Unconditional `goto` skips to the "else" part if the condition fails.

(ii) While Loop
High-level code:

```c
while (a < b)
{
    a = a + 1;
}
```

TAC Representation:

```vbnet
L1: if a >= b goto L2
    a = a + 1
    goto L1
L2:
```

Explanation:

- The loop condition is checked at the top (L1).
- The body executes and jumps back until condition fails.

(iii) For Loop
High-level code:

```c
for (i = 0; i < n; i = i + 1)
    sum = sum + i;
```

TAC Representation:

```vbnet
i = 0
L1: if i >= n goto L2
    sum = sum + i
    i = i + 1
    goto L1
L2:
```

Explanation:

- Initialization, condition, update, and loop body are explicitly shown with gotos.

(iv) Do-While Loop

High-level code:

```c
do {
    a = a + 1;
} while (a < b);
```

TAC Representation:

```vbnet
L1: a = a + 1
    if a < b goto L1
```

Advantages of TAC:

- **Simple and machine-independent representation**
- **Useful for optimization (common subexpression elimination, dead code removal)**
- **Provides easy translation to assembly/machine code**

(b) Directed Acyclic Graph (DAG)

Introduction:

A Directed Acyclic Graph (DAG) is a data structure used by compilers to represent expressions and basic blocks.
It helps to identify common subexpressions, optimize computations, and eliminate redundant code.

Definition:

A DAG is a graph with directed edges and no cycles, meaning that starting from any node, you cannot return to it again by following the edges.

- **Nodes represent operations or operands.**
- **Edges represent the flow of data (operands → operator → result).**

Purpose in Compiler Design:

- **To represent expressions in Intermediate Code Optimization.**

- **To detect common subexpressions (so they're computed only once).**
- **To determine dependencies between computations.**
- **To eliminate dead code and redundant calculations.**

Example 1:

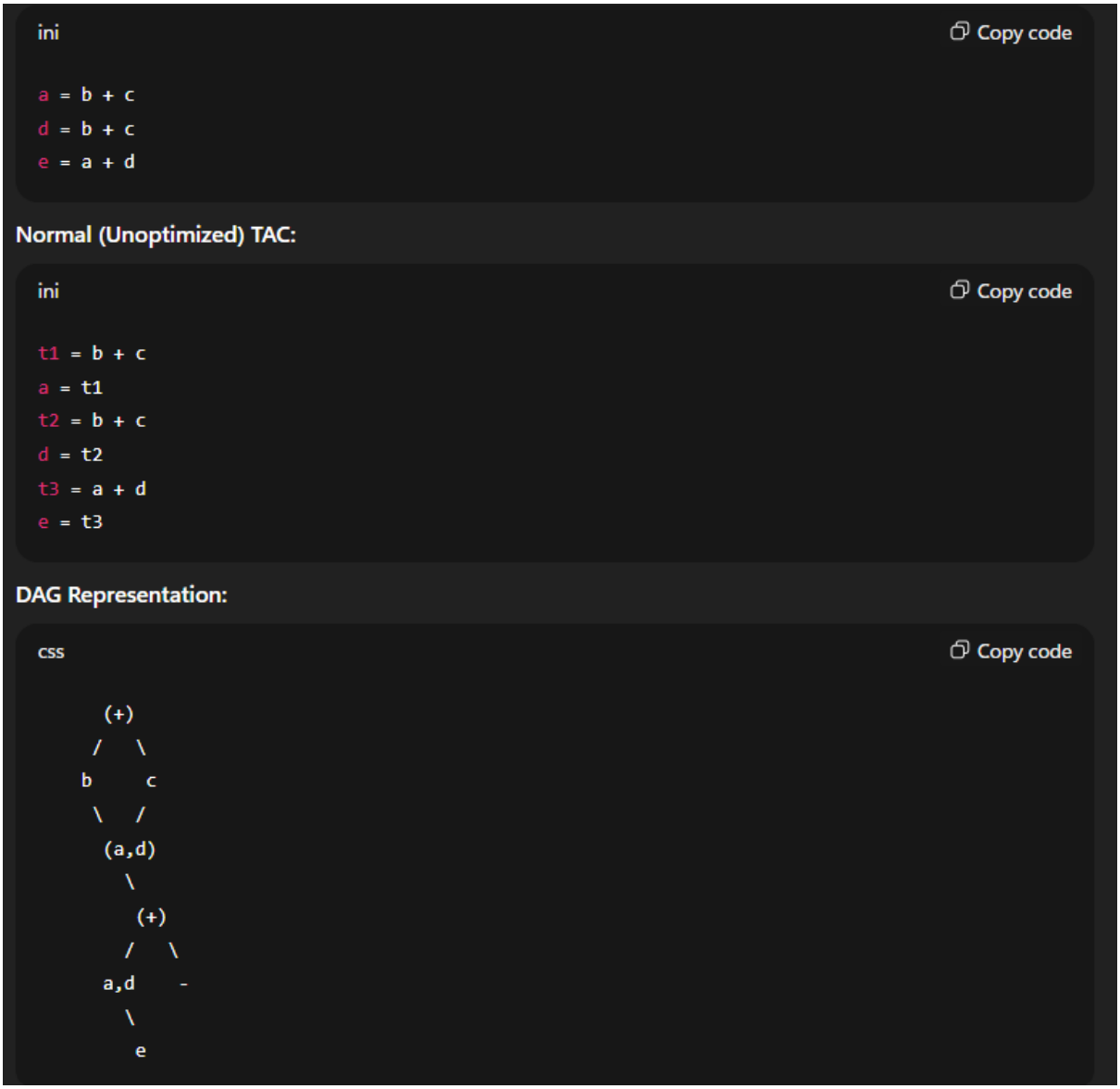
Expression:

```ini
a = b + c
d = b + c
e = a + d
```

Normal (Unoptimized) TAC:

```ini
t1 = b + c
a = t1
t2 = b + c
d = t2
t3 = a + d
e = t3
```

DAG Representation:



Optimization:
Since b + c is computed twice, the DAG identifies it as a common subexpression — so it's calculated only once.

Example 2:

Expression:

```ini
x = (a + b) - (a + b)
```

DAG Representation:

```css
     (-)
    /   \
   (+)  (+)
  / \   / \
 a  b  a  b
```

→ The compiler detects both `(a + b)` as identical subexpressions, eliminates redundancy, and optimizes to:

```ini
x = 0
```

Properties of DAG:

1. **Directed: Edges have direction (operand → operation).**
2. **Acyclic: No loops; ensures evaluation order.**
3. **Common Subexpression Representation: A single node may represent multiple variables.**
4. **Efficient Representation: Reduces redundant calculations.**

Steps to Construct a DAG for a Basic Block:

1. **Identify Expressions in the basic block.**
2. **Check for Existing Nodes:**
   If the same operation with the same operands exists, reuse it.
3. **Create New Node:**
   If no identical node exists, create a new one.
4. **Label Nodes:**
   Assign temporary or variable names to output nodes.

Advantages of DAG:

- **Eliminates common subexpressions.**
- **Removes dead code.**
- **Improves execution efficiency.**
- **Provides clear data dependency among computations.**

Q10. What is Operator Precedence Parsing? Give an Example

---

1. Introduction

Operator Precedence Parsing is a type of bottom-up parsing technique used by compilers to analyze expressions involving operators and operands, such as arithmetic and logical expressions.

It is particularly useful for grammars that describe expressions with operators of different precedence and associativity, like +, -, *, /, ^, etc.

This parser uses the relative precedence (priority) and associativity of operators to decide:

- **When to shift (read the next input symbol)**
- **When to reduce (apply a grammar production)**

---

2. Basic Idea

Instead of using a complex LR parsing table, the operator precedence parser uses a simpler operator precedence relation table, which defines how two terminals (operators or operands) relate to each other.

These relations are:

- **<· → "less than" (the operator on the left has lower precedence)**
- **=· → "equal to" (used for matching parentheses)**
- **>· → "greater than" (the operator on the left has higher precedence)**

Example of Relations:

Operator Pair	Precedence Relation
+ before *	<· (shift)
* before +	>· (reduce)
(before +	<· (shift)
) after (	=· (match)

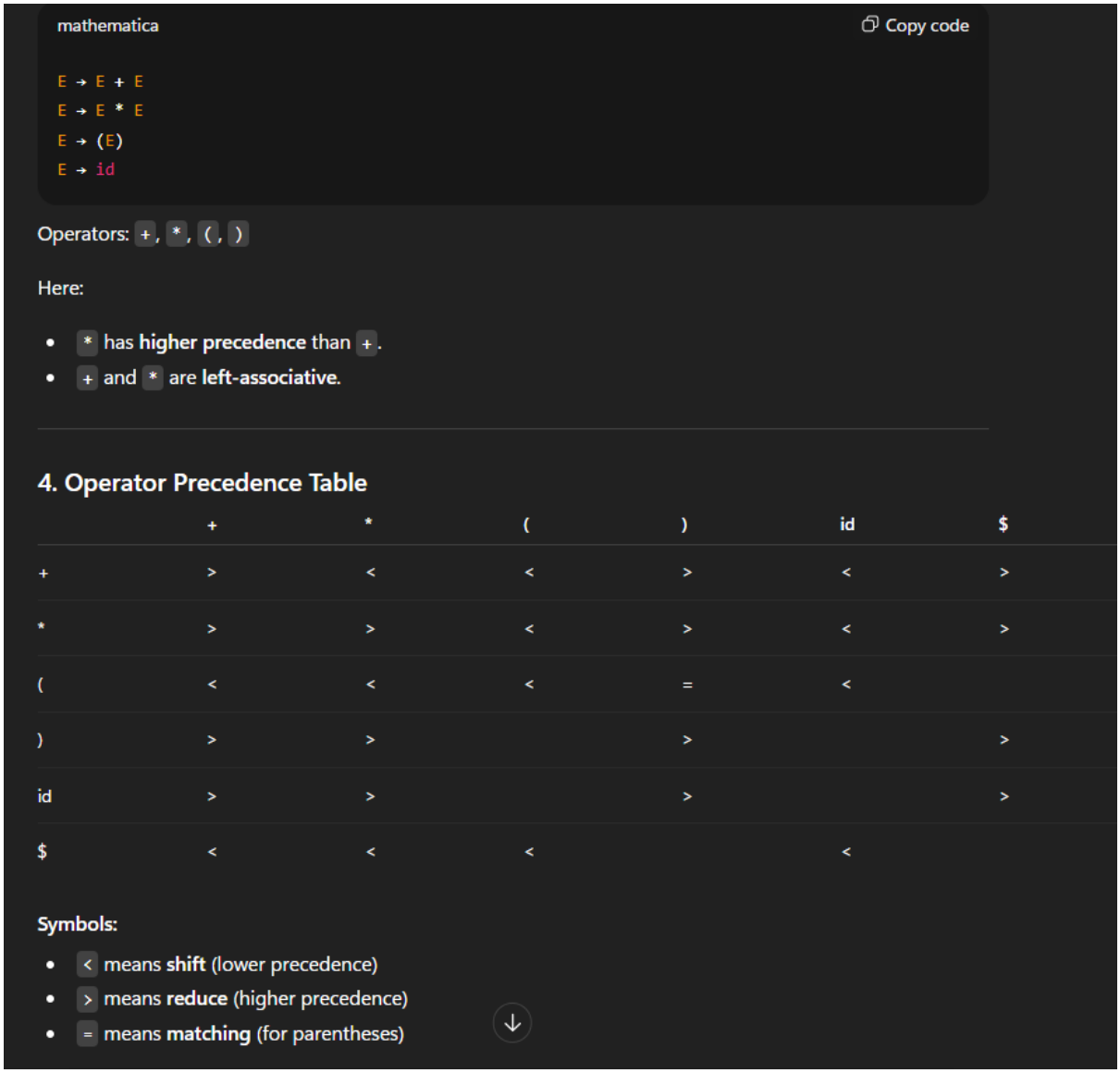
---

3. Grammar Suitable for Operator Precedence Parsing

Only Operator Precedence Grammars can be parsed using this method.
A grammar is operator-precedence if:

1. **No ε-productions (no empty rules).**
2. **No two adjacent non-terminals.**
3. **The grammar defines precedence and associativity clearly.**

Example Grammar:



5. Parsing Process (Algorithm)

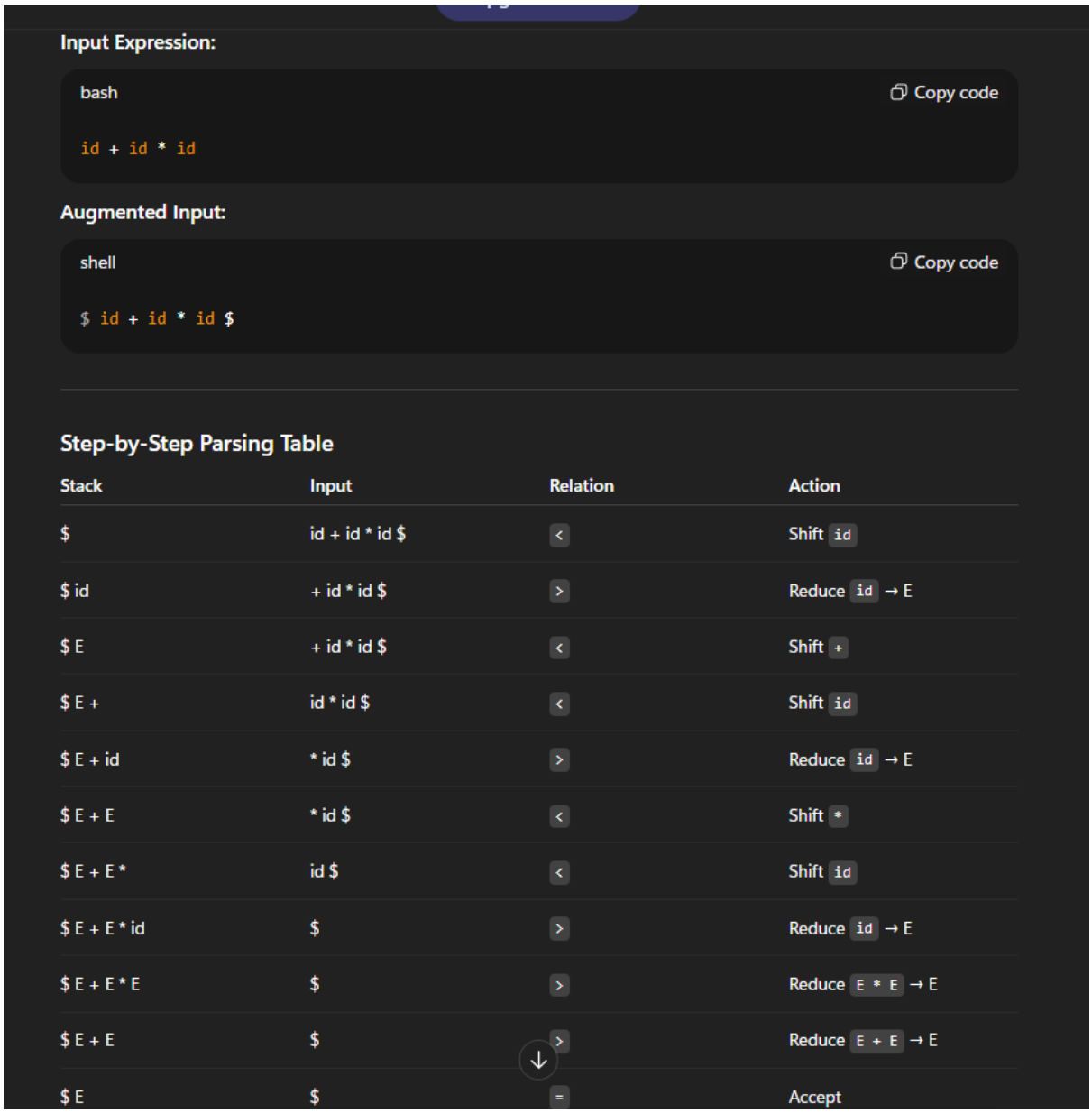
Step 1: Push $ (end marker) onto the stack and read the first input symbol.
Step 2: Compare precedence of top terminal of the stack with current input symbol.
Step 3:

- **If relation is <· or =· → Shift (push input symbol onto stack).**

- **If relation is >· → Reduce (pop until <· is found).**
- **If no relation → Error.**
  Step 4: Repeat until both stack and input contain $.

---

6. Example



Input Expression:

bash — Copy code

```
id + id * id
```

Augmented Input:

shell — Copy code

```
$ id + id * id $
```

Step-by-Step Parsing Table

Stack	Input	Relation	Action
$	id + id * id $	<	Shift id
$ id	+ id * id $	>	Reduce id → E
$ E	+ id * id $	<	Shift +
$ E +	id * id $	<	Shift id
$ E + id	* id $	>	Reduce id → E
$ E + E	* id $	<	Shift *
$ E + E *	id $	<	Shift id
$ E + E * id	$	>	Reduce id → E
$ E + E * E	$	>	Reduce E * E → E
$ E + E	$	>	Reduce E + E → E
$ E	$	=	Accept

7. Advantages of Operator Precedence Parsing

1. **Simple implementation: No need for complex parsing tables like LR parsers.**
2. **Efficient for arithmetic expressions: Works very well for expressions with operators and parentheses.**
3. **No backtracking required.**
4. **Error detection: Errors are detected when precedence relations are missing or invalid.**

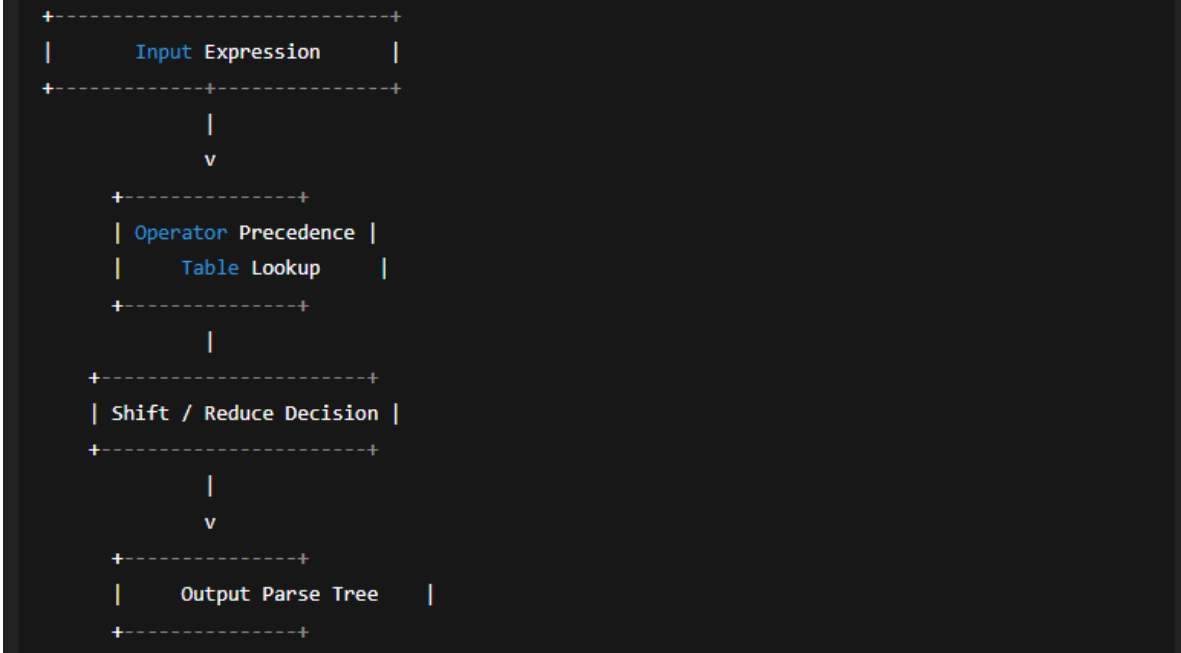
8. Limitations

1. **Cannot handle all context-free grammars.**
2. **Not suitable for constructs like nested if-else or complex language structures.**
3. **Works only for operator precedence grammars.**
4. **Hard to handle unary operators like -x properly without modification.**

9. Diagram (Text Representation)

pgsql

Copy code



Error Handling in LL Parsers (14 Marks)

---

1. Introduction

In any compiler, error handling is an essential part of the syntax analysis (parsing) phase. During parsing, the compiler detects syntax errors that occur due to violations of the grammar rules of a programming language.

An LL parser, which is a top-down parser, reads the input from Left to right and constructs the Leftmost derivation of the sentence. While parsing, if the parser finds an unexpected symbol that does not match the expected grammar rule, it must handle the error in such a way that:

- **The error is reported clearly,**
- **The parser recovers and continues parsing the remaining input, and**
- **It prevents false cascading errors (errors that occur because of a previous one).**

---

2. Need for Error Handling in LL Parsers

Without proper error handling, a single missing semicolon or incorrect keyword can cause the parser to stop immediately, making it difficult for programmers to identify and fix multiple errors. Error handling ensures that the parser continues after an error, giving useful diagnostic messages to the programmer.

---

3. Types of Errors

Before understanding how LL parsers handle errors, it's important to know what types of errors can occur during compilation:

1. **Lexical Errors: Occur during token recognition (e.g., invalid identifier like 123abc).**
2. **Syntax Errors: Occur when input violates grammar rules (e.g., missing semicolon, unmatched parenthesis).**
3. **Semantic Errors: Occur when syntactically correct statements make no sense (e.g., using an undeclared variable).**
4. **Logical Errors: Occur when the output is not as expected even though syntax and semantics are correct.**

LL parsers mainly deal with syntax errors.

---

4. Error Handling Strategies in LL Parsers

There are several methods used for error recovery in LL parsers:

a) Panic Mode Recovery

- **In this method, when an error is detected, the parser discards symbols from the input until a symbol in a synchronizing set is found.**

- **The synchronizing set may include keywords or delimiters such as ;, }, end, etc.**
- **This allows the parser to skip erroneous parts and resume parsing from a stable point.**

Example:
If a missing semicolon is found in a C program, the parser can discard tokens until it finds the next semicolon ; or closing brace }.

Advantages:

- **Simple to implement.**
- **Ensures parser does not enter infinite loops.**
  Disadvantages:
- **May skip large portions of input, losing potential valid information.**

---

b) Phrase-Level Recovery

- **Here, the parser performs local correction at the point of error to repair the input.**
- **For example, it might insert a missing symbol, delete an extra token, or replace an incorrect one.**
- **The idea is to make a small local change to continue parsing.**

Example:
If the parser expects) but finds ;, it might insert the missing) automatically.

Advantages:

- **Provides more meaningful error messages.**
  Disadvantages:
- **Difficult to implement correctly; might introduce incorrect repairs.**

---

c) Error Productions

- **In this method, special error rules are added to the grammar.**
- **These rules describe common mistakes made by programmers.**
- **When the parser encounters such patterns, it applies the error production rule and prints a helpful message.**

Example:
If an expression grammar expects E → E + T, but user writes E + + T, then a rule like E → E error T can handle the case.

---

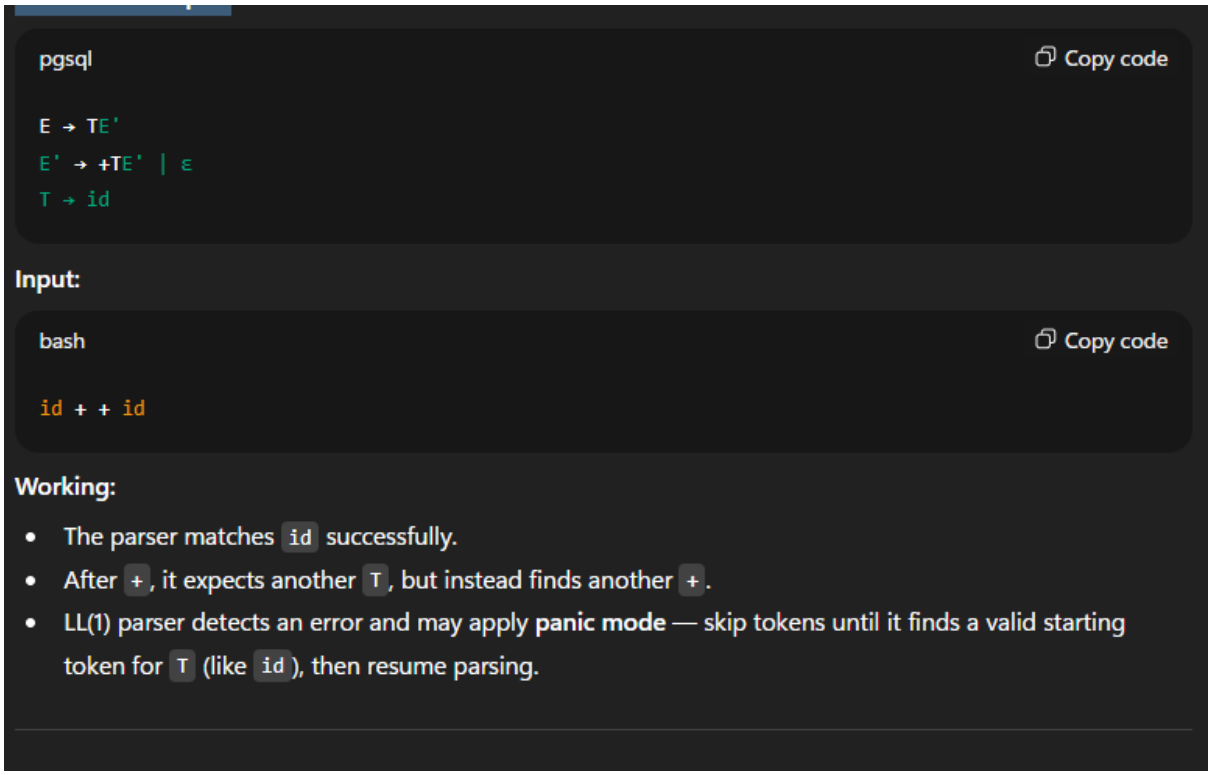
d) Global Correction

- **The parser tries to make the minimum number of changes (insertions, deletions, or replacements) in the input string so that it becomes a valid program according to grammar rules.**
- **Although accurate, it is computationally expensive and rarely used in practical compilers.**

---

5. Example of LL(1) Parser Error Handling

Grammar Example:



Syntax-Directed Definitions (SDD)

---

1. Introduction

A Syntax-Directed Definition (SDD) is a formal method used in compilers to specify the translation of programming language constructs.
It associates semantic rules (actions) with the grammar rules of a language.
These semantic rules describe how to compute attributes of grammar symbols (both terminals and non-terminals).

In simple words, SDD links syntax (structure) and semantics (meaning) of a program.

---

2. Definition

A Syntax-Directed Definition is a context-free grammar (CFG) where each grammar symbol is associated with a set of attributes, and each production rule has a set of semantic rules for computing the attribute values.

---

3. Purpose of Syntax-Directed Definitions

- **To specify the translation of source code into intermediate code, target code, or other forms.**
- **To perform semantic analysis (type checking, scope resolution, etc.).**
- **To help construct syntax trees or intermediate representations during compilation.**
- **To define how values and types are computed during parsing.**

---

4. Components of SDD

1. **Grammar Symbols:**
   These are the terminals (tokens like id, num, +, *) and non-terminals (syntactic variables like E, T, F).
2. **Attributes:**
   Each symbol has attributes that hold information.
   - **Synthesized Attributes: Computed from child nodes in the parse tree and passed upward.**
   - **Inherited Attributes: Computed from parent or sibling nodes and passed downward.**
3. **Semantic Rules:**
   These are equations or actions attached to grammar productions to compute attributes.

---

5. Example of SDD

Consider an expression grammar:

```
r
```

Copy code

```
E → E1 + T
E → T
T → num
```

Semantic rules to evaluate arithmetic expressions:

```
ini
```

Copy code

```
E.val = E1.val + T.val
E.val = T.val
T.val = num.lexval
```

Here:

- `val` is a **synthesized attribute** (computed from children).
- The SDD specifies how the **value of an expression** is calculated.

---

6. Types of Syntax-Directed Definitions

There are mainly two types of SDDs:

1. **S-Attributed Definitions**
2. **L-Attributed Definitions**

A. S-Attributed Definitions

Definition

An S-Attributed Definition is an SDD that uses only synthesized attributes.

This means all attribute values are computed from child nodes and propagate upward in the parse tree.

It is most suitable for bottom-up parsing (like LR parsers) because synthesized attributes are computed naturally in that order.

Example (S-Attributed)

Grammar:

```
r

E → E1 + T
E → T
T → num
```

Semantic rules:

```
ini

E.val = E1.val + T.val
E.val = T.val
T.val = num.lexval
```

Explanation:

- **num.lexval gives the numeric value from the token.**
- **T.val is computed from the terminal.**
- **E.val is computed from the values of its children.**
- **All attributes are synthesized (computed from children).**

Parse Tree Example:

E

/|

E1 + T

- **If E1.val = 2 and T.val = 3, then E.val = 2 + 3 = 5.**

Advantages:

- **Simple and easy to implement.**
- **Works perfectly with bottom-up parsers (e.g., LR parser).**

Disadvantages:

- **Cannot handle cases where information must be passed from parent to child.**

B. L-Attributed Definitions

Definition

An L-Attributed Definition is an SDD that can use both inherited and synthesized attributes, but with restrictions to ensure left-to-right evaluation.

In this type:

- **Inherited attributes are passed from parent to child or from left sibling to right sibling.**
- **Synthesized attributes are still passed upward.**

It is suitable for top-down parsing (like LL parsers).

Example (L-Attributed)

Grammar:

```mathematica
S → L = E
L → id
E → L
```

Attributes:

- `L.name` is an inherited attribute (variable name).
- `E.val` is a synthesized attribute (expression value).

Semantic Rules:

```mathematica
S → L = E       { print("Assign", L.name, "=", E.val) }
L → id          { L.name = id.lexeme }
E → L           { E.val = getValue(L.name) }
```

Explanation:

- **L.name is inherited by E from the left-hand side of the assignment.**
- **When parsing x = y, the value of y is fetched using getValue(L.name) and assigned to x.**
- **Both synthesized and inherited attributes work together.**

Restrictions for L-Attributed SDD:

To be L-Attributed:

1. **Each inherited attribute of a symbol depends only on:**
    - **Attributes of the parent.**
    - **Attributes of symbols to its left in the production.**
2. **Synthesized attributes may depend on all child attributes.**

Difference Between S-Attributed and L-Attributed

Feature	S-Attributed	L-Attributed
Attributes Used	Only Synthesized	Both Synthesized & Inherited
Information Flow	Bottom-up (Child → Parent)	Top-down and Bottom-up
Suitable For	Bottom-up parsers (LR)	Top-down parsers (LL)
Complexity	Simple	Slightly Complex
Example Use	Expression evaluation	Type checking, variable declarations

---

7. Practical Applications

- **Used in syntax-directed translation schemes (SDT).**
- **Helps generate intermediate code, syntax trees, and type information.**
- **Plays a crucial role in semantic analysis and code generation.**

Que 14. Write TAC for the following statements:

a) **x = a + b * c**
b) **if(a < b) then x=y+z**

- # 1. Introduction to Three-Address Code (TAC)
- **Three-Address Code (TAC)** is a type of **Intermediate Representation (IR)** used by compilers between the source code and target machine code.
- It breaks complex high-level language statements into a sequence of **simple instructions**, each containing **at most three addresses (operands)**.
- A typical TAC instruction takes one of the following forms:

```
x = y op z        → binary operation
x = op y          → unary operation
goto L             → unconditional jump
if x relop y goto L  → conditional jump
param x            → passing parameter
call p, n          → procedure call
return x           → return value
```

Each instruction performs only one operation, making it easy to optimize and translate to assembly code.

---

2. Structure of TAC

Each TAC instruction typically includes:

- **Two operands (source values)**
- **One result variable**
- **An operator (+, -, *, /, <, >, etc.)**

Example:

t1 = a + b

x = t1 * c

Here, t1 and x are temporary variables created by the compiler to store intermediate results.

---

3. Rules for Writing TAC

When converting high-level statements into TAC:

1. **Evaluate expressions step by step from lowest to highest precedence.**
2. **Introduce temporary variables for intermediate results.**
3. **Use conditional and unconditional jumps for control structures like if, while, etc.**
4. **Each TAC instruction should have only one operator.**

(a) Statement: x = a + b * c

Step 1: Identify Operators and Precedence

Expression:

x = a + b * c

Operator precedence:

1. *** (multiplication)**
2. **+ (addition)**

So, evaluate b * c first, then a + (b * c).

Step 2: Generate Three-Address Code

Step	TAC Instruction	Explanation
1	t1 = b * c	Multiply b and c, store in t1
2	t2 = a + t1	Add a and t1, store in t2
3	x = t2	Assign the result to x

✅ **Final TAC:**

t1 = b * c

t2 = a + t1

x = t2

Step 3: Explanation

- **Temporary variable t1 holds the result of b * c.**
- **Temporary variable t2 adds a to t1.**
- **The final result is stored in x.**

Thus, the TAC executes the expression step-by-step, following operator precedence.

(b) Statement: if (a < b) then x = y + z

Step 1: Identify Components

Here we have:

- **Condition: (a < b)**
- **Action: x = y + z**

We need to evaluate the condition and, if true, perform the assignment.

Step 2: Generate Three-Address Code

Step	TAC Instruction	Explanation
1	if a < b goto L1	If condition is true, go to label L1
2	goto L2	Else, skip the statement
3	L1: t1 = y + z	Label L1, compute y + z
4	x = t1	Assign the result to x
5	L2:	End of if-statement

✅ **Final TAC:**

```
if a < b goto L1
goto L2
L1: t1 = y + z
  x = t1
L2:
```

Step 3: Explanation

- **The compiler generates a conditional jump using the relational operator <.**
- **If the condition a < b is true, control jumps to label L1.**
- **At L1, the statement x = y + z is executed.**
- **Otherwise, control goes directly to L2, skipping the assignment.**
- **Labels (L1, L2) are used to mark the flow of control.**

---

4. Summary Table

Statement	Three Address Code (TAC)
x = a + b * c	t1 = b * c t2 = a + t1 x = t2

Statement	Three Address Code (TAC)
if (a < b) then x = y + z	if a < b goto L1 goto L2 L1: t1 = y + z x = t1 L2:

---

5. Key Points

- **Each TAC instruction performs a single, simple operation.**
- **Temporary variables (t1, t2) store intermediate results.**
- **Labels (L1, L2) control conditional branching.**
- **TAC helps in code optimization and machine code generation later in the compiler pipeline.**