

Mohsin Iban Hossain

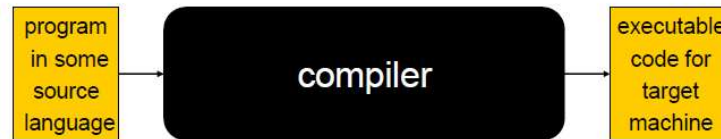
AIUB, Compiler Design Notes

COMPILER DESIGN

INTRODUCTION TO COMPILERS

What is a Compiler?

- ⇒ A **compiler** is a software program that translates **source code** written in a high-level programming language (like C, C++, Java) into machine code (low-level language) that the computer's hardware can understand and execute.

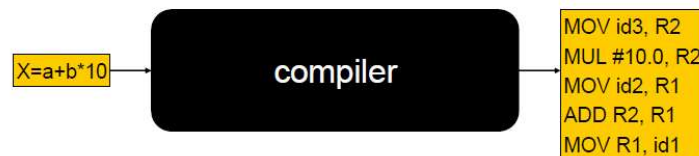


Traditionally, compilers go from high-level languages to low-level languages.

Why Use a Compiler?

- To **convert human-readable code** to machine-readable format.
- To **optimize** code for better performance.
- To **detect errors** in code during compilation (syntax and semantic errors).

❖ Example:



Interpreter

- ⇒ An **interpreter** is a program that **executes instructions written in a programming language line-by-line**, without converting the entire code to machine language at once like a compiler does.

How It Works:

1. **Reads one line of code**
2. **Translates and executes it immediately**
3. **Moves to the next line**

No intermediate machine code is generated or stored permanently.



Differences between Compiler and Interpreter

Feature	Compiler	Interpreter
Translation Time	Translates entire code at once	Translates line by line
Execution Speed	Faster execution	Slower execution
Error Detection	Shows all errors after compilation	Stops at the first error
Example Languages	C, C++, Java (compiled to bytecode)	Python, JavaScript, Ruby

Preprocessor

⇒ A **preprocessor** is a tool that **processes the source code before compilation begins**. It handles **preprocessing directives**, which are instructions for the compiler to perform certain operations like including files or defining constants.

Role of a Preprocessor in Compilation:

- Acts as the **first step** in the compilation process.
- Modifies the source code to produce an **expanded source code** that the compiler uses.
- Operates before the actual compiler phases (like lexical analysis).
- Ensures that the compiler receives clean, ready-to-compile code.

❖ Example:

```
#define PI 3.1416
area = PI * r * r;
```

Preprocessor Directives in C/C++:

Directive	Purpose
<code>#include</code>	Inserts contents of another file
<code>#define</code>	Defines macros or constants
<code>#undef</code>	Undefines a macro
<code>#ifdef/#ifndef</code>	Conditional compilation
<code>#if, #else, #elif, #endif</code>	More conditional logic
<code>#pragma</code>	Compiler-specific instructions

Assembler

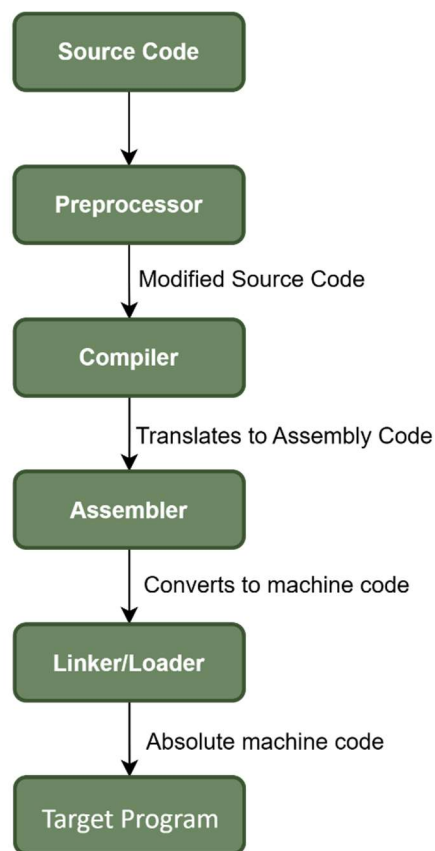
- ⇒ An **assembler** is a program that **translates assembly language code into machine code** (binary format). It serves as a bridge between **low-level human-readable assembly language** and **machine-executable instructions**.



Language Processing System

- ⇒ A **Language Processing System** is a collection of tools/programs that **translate and execute source code** written in a high-level language. It includes **compiler, assembler, linker, and loader**.

Flow of Language Processing System



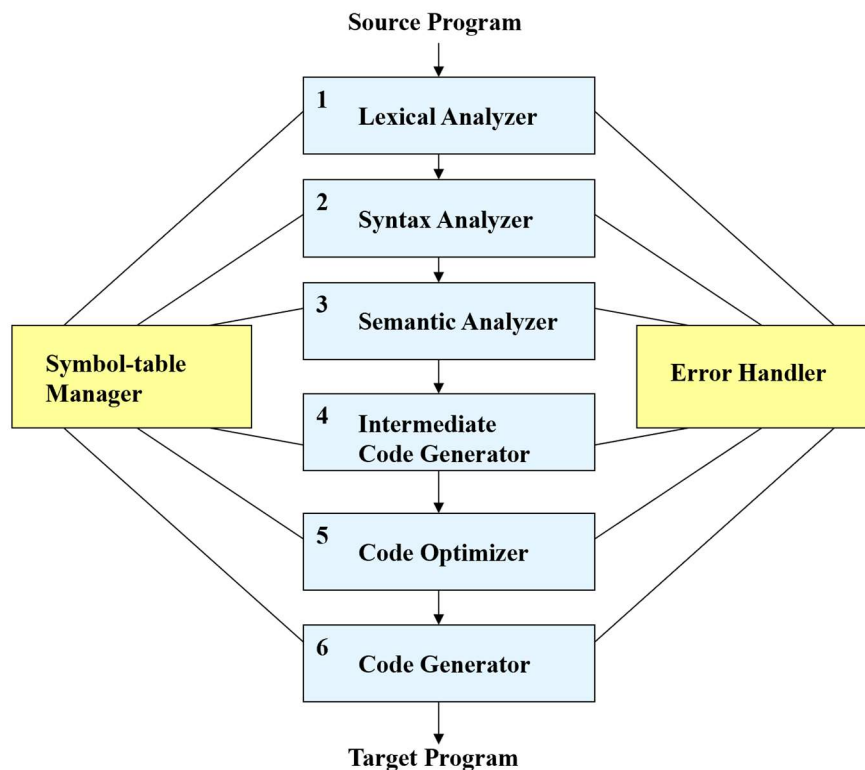
Phases of Language Processing

- **Editor** → Used to write the source program.
- **Preprocessor** → Handles macros, file inclusion, etc.
- **Compiler** → Translates high-level code to assembly.
- **Assembler** → Converts assembly code to machine code.
- **Linker** → Links object files and libraries to make a single executable.
- **Loader** → Loads the executable into memory and starts execution.

Summary Table:

Phase	Output	Role
Preprocessor	Modified Source	Handles macros, includes
Compiler	Assembly Code	Translates to low-level code
Assembler	Object Code	Converts to machine code
Linker	Executable File	Combines code and libraries
Loader	Running Program	Loads into memory for execution

The Phases of a Compiler



What Are Compiler Phases?

⇒ A **compiler** works in **multiple phases**, each performing a specific task to convert high-level source code into machine code.

➤ These phases are grouped into **two main parts**:

- **Analysis Phase** – Front-end (understanding the source code)
- **Synthesis Phase** – Back-end (generating target code)

Analysis Phase

⇒ It **reads** the source program and **breaks it down** to understand its structure and meaning. It performs **error detection** and builds intermediate code.

- I. Lexical Analyzer
- II. Syntax Analyzer
- III. Semantic Analyzer
- IV. Intermediate code generator

Synthesis Phase

⇒ It takes the analyzed (intermediate) representation of the code and **produces optimized machine-level code** that can be executed.

- V. Code optimizer
- VI. Code generator

Lexical Analyzer (Scanner)

⇒ The **Lexical Analyzer** is the **first phase** of the compiler. It reads the **source code character by character**, groups them into **tokens**, and removes unnecessary elements like **whitespaces** and **comments**.

It acts as a **scanner** that prepares input for the next phase: **Syntax Analysis**.

Main Tasks:

- **Tokenization** – Converts characters into tokens
- **Removes Comments and Whitespaces**
- **Error Reporting** – Reports **lexical (invalid character) errors**
- **Symbol Table Entry** – Adds identifiers and literals to the symbol table

What is a Token?

⇒ A **token** is the smallest meaningful unit in a program.

Token Type	Example
Keyword	if, while, int
Identifier	x, sum, total
Operator	+, -, ==
Separator	;, {, }
Literals	100, 'a', "text"

❖ Example1:

```
int x = 5;
```

Tokens:

```
int → KEYWORD
x → IDENTIFIER
= → OPERATOR
5 → INTEGER_LITERAL
; → SEPARATOR
```

❖ Example2:

```
position = initial + rate * 60
```

The Lexical Analyzer produces:	Final Output:																
<table> <tr> <th>Lexeme</th><th>Token</th></tr> <tr> <td>position</td><td>(id, 1)</td></tr> <tr> <td>=</td><td>(=)</td></tr> <tr> <td>initial</td><td>(id, 2)</td></tr> <tr> <td>+</td><td>(+)</td></tr> <tr> <td>rate</td><td>(id, 3)</td></tr> <tr> <td>*</td><td>(*)</td></tr> <tr> <td>60</td><td>(60)</td></tr> </table>	Lexeme	Token	position	(id, 1)	=	(=)	initial	(id, 2)	+	(+)	rate	(id, 3)	*	(*)	60	(60)	<pre> position = initial + rate * 60 ↓ ↓ ↓ ↓ ↓ ↓ <id,1> <=> <id,2> <+> <id,3> <*> <60></pre>
Lexeme	Token																
position	(id, 1)																
=	(=)																
initial	(id, 2)																
+	(+)																
rate	(id, 3)																
*	(*)																
60	(60)																

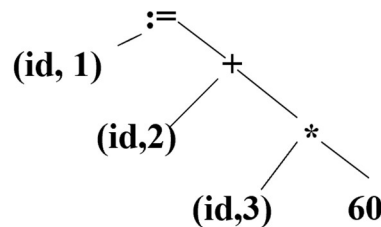
Syntax Analyzer

- ⇒ The **Syntax Analyzer**, also known as the **parser**, is the **second phase** of the compiler. It takes the stream of **tokens** from the lexical analyzer and checks whether the sequence follows the **grammar rules** of the programming language.

Main Tasks:

- Checks the **syntactic structure** of the source code
- Builds a **syntax tree**
- Detects **syntax errors**

Syntax Tree:



Syntax Errors:

- ⇒ A **syntax error** is a **violation of language grammar**.
1. Missing brackets, ;, or keywords
 2. Incorrect nesting or order
 3. Example: `if (x > 5 { } → missing)`

Semantic Analyzer

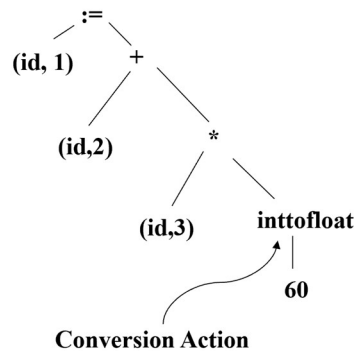
- ⇒ The **Semantic Analyzer** is the **third phase** of the compiler. It checks whether the **meaning** of the program is **correct** by applying the rules of the language's **semantics**.

It ensures that statements make **logical sense**, even if they are **syntactically correct**.

Main Tasks:

- **Type checking** (e.g., can't add **int** to **string**)
- **Checks for undeclared variables**
- **Validates function calls** (e.g., correct number and type of arguments)
- Ensures variables are used according to their declared type
- Adds information to the **symbol table**

Annotated syntax tree:



Semantic Errors:

Type	Example
Type mismatch	<code>int x = "hello";</code>
Undeclared variable	<code>y = 5;</code> (if <code>y</code> is not declared)
Invalid operation	<code>true + 1</code>
Wrong number of arguments	<code>sum(2)</code> if <code>sum(int, int)</code>

Intermediate code generator

⇒ After semantic analysis, the **Intermediate Code Generator** translates the source program into a simpler **intermediate representation (IR)** — a form that is **easier to analyze and optimize** before generating final machine code.

It acts as a **bridge** between the front-end (analysis) and back-end (synthesis) of the compiler.

Key Properties of Intermediate Code:

1. Easy to produce
2. Easy to translate into machine code

Common Forms of Intermediate Code:

Type	Description	Example
Three-Address Code (TAC)	One operator, at most 3 operands	<code>t1 = a + b</code>
Postfix (Polish) Notation	Operands before operators	<code>ab+</code>
Syntax Trees / DAGs	Graph-based structures	Nodes represent operations

Note: In this course we consider an intermediate form called **“Three Address Code”**

❖ Example1:

Source Code:	Intermediate Code (TAC):
<code>a = b + c * d;</code>	<pre> t1 = c * d t2 = b + t1 a = t2 </pre>

❖ Example2:

For the expression:	Intermediate Code (TAC):
<code>id1 = id2 + id3 * 60</code>	<pre> temp1 = inttofloat(60) temp2 = id3 * temp1 temp3 = id2 + temp2 id1 = temp3 </pre>

Code Optimizer

⇒ The **Code Optimizer** improves the **intermediate code** so that **faster and smaller** target code can be generated **without changing the output**.

Key Properties of Code Optimizer:

- Finds **more efficient ways** to execute code
- **Replaces code** with optimized versions
- **Improves runtime performance**

❖ Optimized Output Example:

From: Intermediate Code (TAC)	To: Code Optimizer
<pre> temp1 = inttofloat(60) temp2 = id3 * temp1 temp3 = id2 + temp2 id1 = temp3 </pre>	<pre> temp1 = id3 * 60.0 id1 = id2 + temp1 </pre>

Code Generator

⇒ The **Code Generator** translates the **optimized intermediate code** into **target machine code** (assembly or binary).

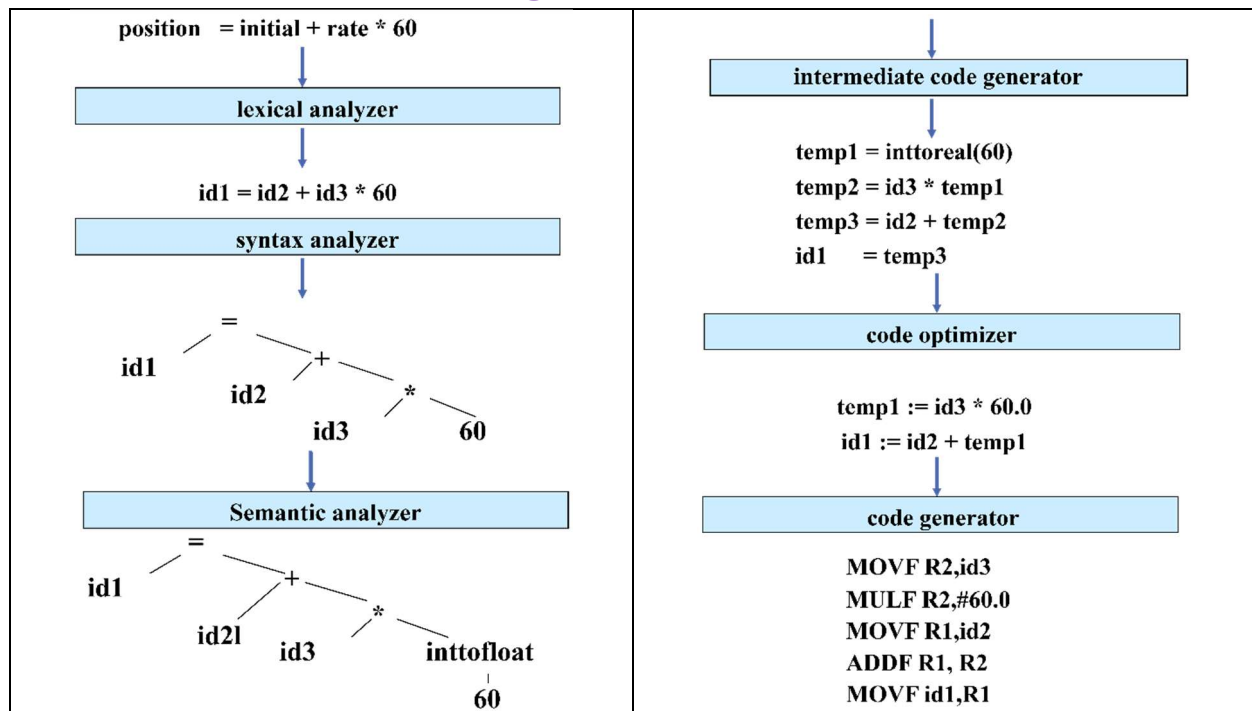
Responsibilities:

- Convert each intermediate instruction into **machine-specific instructions**
- **Allocate registers** efficiently
- Manage **memory addresses** and **variables**
- Generate **correct and efficient** executable code

❖ Example:

From: Optimized Intermediate Code	To: Target Code (Assembly-Like)
<pre>temp1 = id3 * 60.0 id1 = id2 + temp1</pre>	<pre>MOVF R2, id3 MULF R2, #60.0 MOVF R1, id2 ADDF R1, R2 MOVF id1, R1</pre>

Reviewing the Entire Process

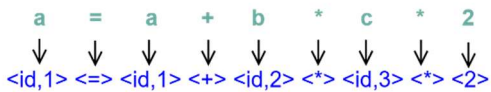
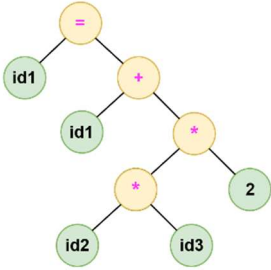
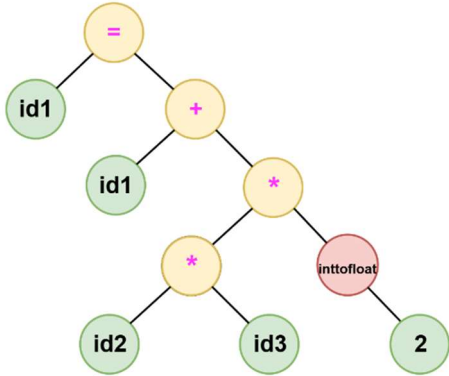


Practice Problem

🔗 **Problem1:** For the following statement show output of different phases of a Compiler, where **a, b, c** are the float type variables.

$$a = a + b * c * 2$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer Taking operator precedence: Multiplication (*) > Addition (+)
 <p>Final Output: $id1 = id1 + (id2 * id3) * 2$</p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<pre> t1 = id2 * id3 t2 = inttofloat(2) t3 = t1 * t2 t4 = id1 + t3 id1 = t4 </pre>
Step5: Code Optimizer	Step6: Code Generator
<pre> t1 = id2 * id3 t2 = t1 * 2.0 id1 = id1 + t2 </pre>	<pre> MOVF R2,id2 MOVF R3,id3 MULF R2,R3 MULF R2,#2.0 MOVF R1,id1 ADDF R1,R2 MOVF id1,R1 </pre>

🔗 **Problem2:** For the following statement show output of different phases of a Compiler, where **y, b, c, d** are the float type variables.

$$y = b + c - d + 20$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer
<p> $y = b + c - d + 20$ $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$ $\langle id,1 \rangle \langle == \rangle \langle id,2 \rangle \langle + \rangle \langle id,3 \rangle \langle - \rangle \langle id,4 \rangle \langle + \rangle \langle 20 \rangle$ Final Output: $id1 = id2 + id3 - id4 + 20$ </p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<p> $t1 = id2 + id3$ $t2 = t1 - id4$ $t3 = inttofloat(20)$ $t4 = t2 + t3$ $id1 = t4$ </p>
Step5: Code Optimizer	Step6: Code Generator
<p> $t1 = id2 + id3$ $t2 = t1 - id4$ $id1 = t2 + 20.0$ </p>	<p> $MOV\ F\ R2, id2$ $MOV\ F\ R3, id3$ $ADD\ F\ R2, R3$ $MOV\ F\ R4, id4$ $SUB\ F\ R2, R4$ $ADD\ F\ R2, \#20.0$ $MOV\ F\ id1, R2$ </p>

🔗 **Problem3:** For the following statement show output of different phases of a Compiler, where **p, s, u, v, w** are the float type variables.

$$p = 22 * s / (u + v) * w - u$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer
<p> $p = 22 * s / (u + v) * w - u$ $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$ $\langle id, 1 \rangle \langle = \rangle \langle 22 \rangle \langle * \rangle \langle id, 2 \rangle \langle / \rangle \langle (\rangle \langle id, 3 \rangle \langle + \rangle \langle id, 4 \rangle \langle * \rangle \langle id, 5 \rangle \langle - \rangle \langle id, 3 \rangle$ </p> <p>Final Output: $id1 = 22 * id2 / (id3 + id4) * id5 - id3$</p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<pre> t1 = id3 + id4 t2 = inttofloat(20) t3 = t2 * id2 t4 = t3 / t1 t5 = t4 * id5 t6 = t5 - id3 id1 = t6 </pre>
Step5: Code Optimizer	Step6: Code Generator
<pre> t1 = id3 + id4 t2 = 22.0 * id2 t3 = t2 / t1 t4 = t3 * id5 id1 = t4 - id3 </pre>	<pre> MOVF R3, id3 MOVF R4, id4 ADDF R4, R3 MOVF R4, id2 MULF R4, #22.0 DIVF R4, R3 MOVF R5, id5 MULF R4, R5 SUBF R4, R3 MOVF id1, R4 </pre>

🔗 **Problem4:** For the following statement show output of different phases of a Compiler, where **x, a, b, c** are the float type variables.

$$x = a + (b + c)/2 - 30$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer
<pre> x = a + (b + c) / 2 - 30 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ <id,1> <=> <id,2> <+> <(> <id,3> <+> <id,4> <)> </> <2> <-> <30> </pre> <p>Final Output: $id1 = id2 + (id3 + id4) / 2 - 30$</p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<pre> t1 = id3 + id4 t2 = inttofloat(2) t3 = t1/t2 t4 = id2 + t3 t5 = inttofloat(30) t6 = t4 - t5 id1 = t6 </pre>
Step5: Code Optimizer	Step6: Code Generator
<pre> t1 = id3 + id4 t2 = t1/2.0 t3 = id2 + t2 id1 = t3 - 30.0 </pre>	<pre> MOVF R3, id3 MOVF R4, id4 ADDF R3, R4 DIVF R3, #2.0 MOVF R2, id2 ADDF R2, R3 SUBF R2, #30.0 MOVF id1, R2 </pre>

🔗 **Problem5:** For the following statement show output of different phases of a Compiler, where **x, a, b, c, d** are the float type variables.

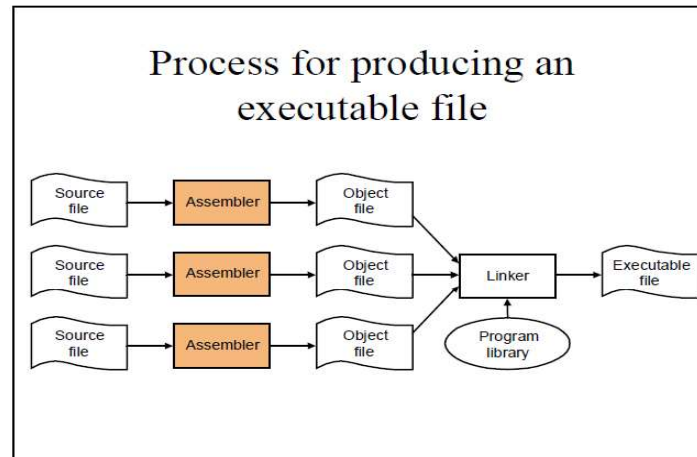
$$x = x + \{(a + b) + 4\}/c - d$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer
<pre> x = x + { (a + b) + 4 } / c - d ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ <id,1> <=> <id,1> <+> <{> <(> <id,2> <+> <id,3> <)> <+> <4> <}> </> <id,4> <-> <id,5> </pre> <p>Final Output: <code>id1 = id1+ {(id2 + id3) + 4} / id4 - id5</code></p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<pre> t1 = id2 + id3 t2 = inttofloat(4) t3 = t1 + t2 t4 = t3/id4 t5 = id1 + t4 t6 = t5 - id5 id1 = t6 </pre>
Step5: Code Optimizer	Step6: Code Generator
<pre> t1 = id2 + id3 t2 = t1 + 4.0 t3 = t2/id4 t4 = id1 + t3 id1 = t4 - id5 </pre>	<pre> MOVF R2, id2 MOVF R3, id3 ADDF R2, R3 ADDF R2, #4.0 MOVF R4, id4 DIVF R2, R4 MOVF R1, id1 ADDF R1, R2 MOVF R5, id5 SUBF R1, R5 MOVF id1, R1 </pre>

Linker

- A **linker** combines object modules into a single executable program.
- Used in **modular programming** to simplify large programs.
- After compilation, it **joins all object files**.
- Also called **link editor** or **binder**.



Loader

- A **loader** copies programs from storage to **main memory**.
- Makes programs **ready for execution**.
- It is a **special type of system program**.

Front end and Back end of a Compiler

➤ Front-End Phases:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation

➤ Back-End Phases:

1. Code Optimization
2. Code Generation

Advantages of Using Front-end and Back- end

Retargeting:

- Build compiler for a **new machine**.
- Use existing **front-end**, attach new **code generator**.

Optimization:

- Reuse **intermediate code optimizers**.
- Works for **different languages** and **machines**.

Symbol Table Management

- A **data structure** storing all identifiers and their **attributes**.
- For **variables**: type, memory size, scope.
- For **functions/procedures**: argument types, passing methods, return type.

Purpose of Symbol Table:

- Provides **quick access** to identifier attributes.
- Built during the **analysis phase**.
- Used during the **synthesis phase** of compilation.

Error Handler

- ⇒ Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:
 - **Detect and report** errors clearly.
 - **Correct** errors if possible.
 - **Continue processing** to find more errors.

✂ **Problem4:** What is the purpose of using symbol table management? Explain the advantages of splitting the phases of the compiler into Front End and Back End

➤ **Purpose of Symbol Table Management:**

⇒ A symbol table stores **information about identifiers** (name, type, scope, memory location).

Purpose of Symbol Table:

- Provides **quick access** to identifier attributes.
- Built during the **analysis phase**.
- Used during the **synthesis phase** of compilation.

➤ **Advantages of Splitting the Compiler into Front-End and Back-End**

⇒ The **compiler is divided** into two main parts:

1. **Front End:** Responsible for analyzing the source code
2. **Back End:** Responsible for generating and optimizing the machine code

➤ **Front-End Phases:**

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation

➤ **Back-End Phases:**

3. Code Optimization
4. Code Generation

➤ **Advantages:**

Retargeting:

- Build compiler for a **new machine**.
- Use existing **front-end**, attach new **code generator**.

Optimization:

- Reuse **intermediate code optimizers**.
- Works for **different languages and machines**.

CONTEXT FREE GRAMMAR (CFG)

Context-Free Grammar

→ A **grammar** consists of:

- **Variables (non-terminals):** Usually written in capital letters. One is the **start symbol**.
- **Terminals:** The actual characters or symbols from the alphabet.
- **Productions (rules):** Define how variables can be replaced with terminals or other variables.
- **Start Symbol:** The **start symbol** tells where to begin generating strings.

Formal Definition

⇒ A context Free Grammar (CFG) is a 4-tuple such that-

$$G = (V, T, P, S)$$

where-

- V = Finite non-empty set of variables / non-terminal symbols
- T/Σ = Finite set of terminal symbols
- P/R = Finite non-empty set of production rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
- S = Start symbol ($S \in V$)

Applications:

- Context Free Grammar (CFG) is of great practical importance. It is used for following purposes-
- For defining programming languages
- For parsing the program by constructing syntax tree
- For translation of programming languages
- For describing arithmetic expressions
- For construction of compilers

Example of CFG

✂ Example 1:

⇒ Construct a CFG for $\{0^n 1^n (n \geq 1)\}$

>> Let's break down the conditions:

- if, $n = 1$, $\rightarrow 0^1 1^1 = \{01\}$
- if, $n = 2$, $\rightarrow 0^2 1^2 = \{0011\}$
- if, $n = 3$, $\rightarrow 0^3 1^3 = \{000111\} \dots$

so, CFG:

$S \rightarrow 01 \mid 0S1$ OR $S \rightarrow 01$
 $S \rightarrow 0S1$

$\therefore V = \{S\}$

$\therefore T/\Sigma = \{0, 1\}$

$\therefore P/R = \{S \rightarrow 01, S \rightarrow 0S1\}$

$\therefore S = \{S\}$

>> Let's Check 000111 string Accepted or Rejected:

$S \rightarrow 0S1$

$S \rightarrow 00S11$

$S \rightarrow 000111$

\therefore Accepted

✂ Example 2:

⇒ Identify the **terminal**, **non-terminal**, **Start variable** for the following grammar.

1. $E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

so, From CFG

$\therefore V = \{E, T, F\}$

$\therefore T/\Sigma = \{+, *, (,), id\}$

$\therefore S = \{E\}$

2. $S \rightarrow (L) \mid a$
 $L \rightarrow L, S \mid S$

so, From CFG

$\therefore V = \{S, L\}$

$\therefore T/\Sigma = \{(,), ,\}$

$\therefore S = \{S\}$

Derivation

⇒ A **grammar** creates strings by:

- Starting with the **start symbol**.
- Repeatedly replacing non-terminals using the **production rules**.

The final strings made only of **terminals** form the **language** of the grammar.

🔗 Example 1:

»Consider the following **context free grammar**. Show how the string **0011** can be **generated** by this grammar.

$$S \rightarrow \varepsilon$$

$$S \rightarrow 0S1$$

» Let's Generated the string 0011:

$$S \rightarrow 0S1$$

$$S \rightarrow 00S11$$

$$S \rightarrow 00\varepsilon11$$

$$S \rightarrow 0011$$

🔗 Example 2:

»Consider the following **context free grammar**. Show how the string **011100** can be **generated** by this grammar.

$$S \rightarrow 0S1S \mid 1S0S \mid \varepsilon$$

» Let's Generated the string 011100:

$$S \rightarrow 0S1S$$

$$S \rightarrow 0\varepsilon11S0S$$

$$S \rightarrow 0\varepsilon111S0S0S$$

$$S \rightarrow 0\varepsilon111\varepsilon0\varepsilon0\varepsilon$$

$$S \rightarrow 011100$$

☞ Example 3:

» This **Context-Free Grammar (CFG)** makes sentences using **noun phrases (NP)** and **verb phrases (VP)**:

$$S \rightarrow NP VP$$

$$NP \rightarrow the N$$

$$VP \rightarrow V NP$$

$$V \rightarrow sings \mid eats$$

$$N \rightarrow cat \mid song \mid canary$$

⇒ It can generate sentences like “**the canary sings the song**”, but also strange ones like “**the song eats the cat**”.

Note: It means the CFG creates **grammatically correct (legal)** sentences, even if they don't make real-world sense (not meaningful).

Practice Problem

☞ Problem1:

» Consider the following **context free grammar**. Show how the string “**{{{({})} (())}**” can be **generated** by this grammar using **left derivations**.

$$S \rightarrow A$$

$$A \rightarrow AB \mid \{B\}$$

$$B \rightarrow (A) \mid ($$

» Let's Generated the string “{{{({})} (())}”:

$$S \rightarrow A$$

$$S \rightarrow \{B\}$$

$$S \rightarrow \{(A)\}$$

$$S \rightarrow \{(AB)\}$$

$$S \rightarrow \{(\{B\}B)\}$$

$$S \rightarrow \{(\{(O\}B)\}$$

$$S \rightarrow \{(\{(O\}O)\}$$

☞ Problem2:

» Give the formal definition of the following context free grammar

$$S \rightarrow 0S1 \mid A$$

$$A \rightarrow 1S0 \mid \varepsilon$$

» We Know

- **V** = Set of **variables (non-terminals)**
- **Σ** = Set of **terminals (alphabet symbols)**
- **R** = Set of **production rules**
- **S** = **Start symbol**

so, From CFG

$$\therefore V = \{S, A\}$$

$$\therefore T/\Sigma = \{0, 1\}$$

$$\therefore S = \{S\}$$

$$\therefore R = S \rightarrow 0S1$$

$$S \rightarrow A$$

$$A \rightarrow 1S0$$

$$A \rightarrow \varepsilon$$

PARSE TREES AND AMBIGUITY

Ambiguous Grammar

⇒ A grammar is said to be ambiguous if for any string generated by it, it produces more than one-

- Parse tree
- Or derivation tree
- Or syntax tree
- Or leftmost derivation
- Or rightmost derivation

✂ Example 1:

⇒ Consider the following grammar-

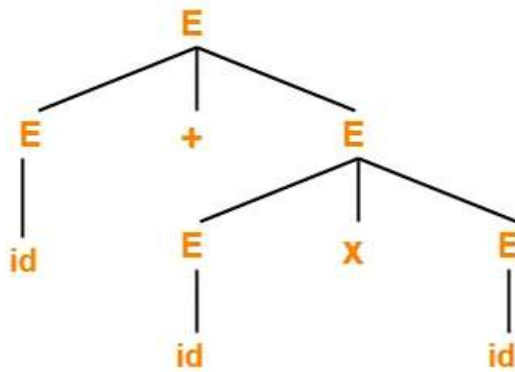
$$E \rightarrow E + E \mid E \times E \mid id$$

Reason-01:

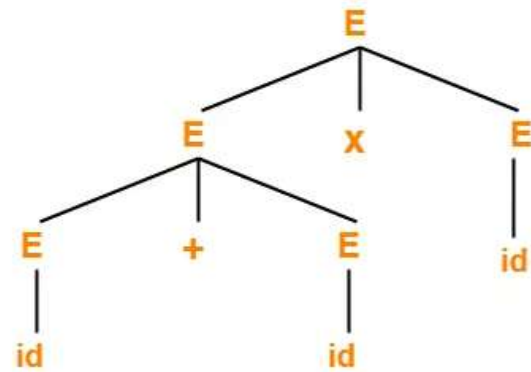
Let us consider a string w generated by the grammar-

$$w = id + id \times id$$

Now, let us draw the parse trees



Parse Tree-01



Parse Tree-02

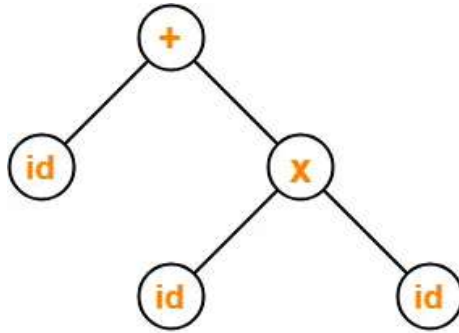
Since two parse trees exist for string w , therefore the grammar is ambiguous.

Reason-02:

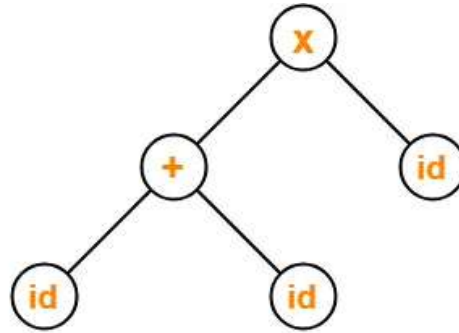
Let us consider a string w generated by the grammar-

$$w = \text{id} + \text{id} \times \text{id}$$

Now, let us draw the syntax trees



Syntax Tree-01



Syntax Tree-02

Since two syntax trees exist for string w , therefore the grammar is ambiguous.

Reason-03:

Let us consider a string w generated by the grammar-

$$w = \text{id} + \text{id} \times \text{id}$$

Now, let us write the leftmost derivations

$E \rightarrow E + E$
 $\rightarrow \text{id} + E$
 $\rightarrow \text{id} + E \times E$
 $\rightarrow \text{id} + \text{id} \times E$
 $\rightarrow \text{id} + \text{id} \times \text{id}$

Leftmost Derivation-01

$E \rightarrow E \times E$
 $\rightarrow E + E \times E$
 $\rightarrow \text{id} + E \times E$
 $\rightarrow \text{id} + \text{id} \times E$
 $\rightarrow \text{id} + \text{id} \times \text{id}$

Leftmost Derivation-02

Since two leftmost derivations exist for string w , therefore the grammar is ambiguous.

Reason-04:

Let us consider a string w generated by the grammar-

$$w = id + id \times id$$

Now, let us write the rightmost derivations

$$E \rightarrow E + E$$

$$\rightarrow E + E \times E$$

$$\rightarrow E + E \times id$$

$$\rightarrow E + id \times id$$

$$\rightarrow id + id \times id$$

Rightmost Derivation-01

$$E \rightarrow E \times E$$

$$\rightarrow E \times id$$

$$\rightarrow E + E \times id$$

$$\rightarrow E + id \times id$$

$$\rightarrow id + id \times id$$

Rightmost Derivation-02

Since two rightmost derivations exist for string w , therefore the grammar is ambiguous.

Problem 1:

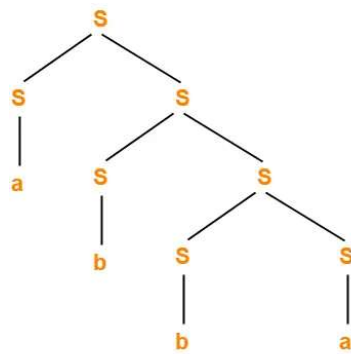
⇒ Check whether the given grammar is ambiguous or not for the following string **abba**

$$S \rightarrow SS \mid a \mid b$$

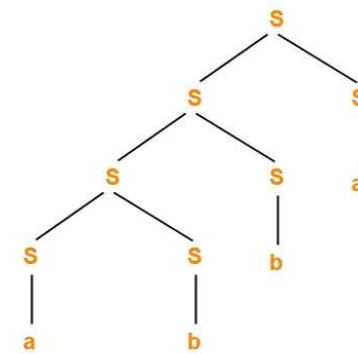
→ is given,

⇒ A string w generated by the given grammar- $w = abba$

Now, draw the parse trees



Parse tree-01



Parse tree-02

Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

⌘ Problem 2:

⇒ Check whether the given grammar is ambiguous or not for the following string **ab**

$$S \rightarrow A \mid B$$

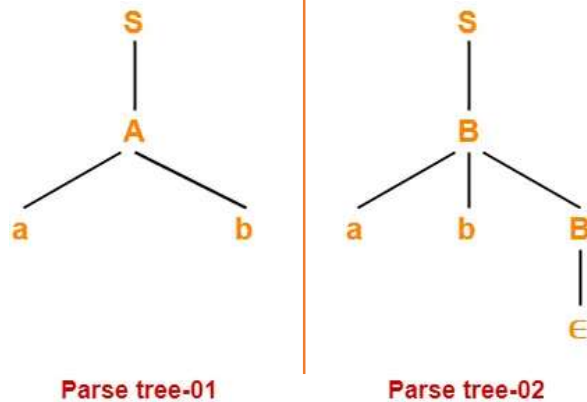
$$A \rightarrow aAb \mid ab$$

$$B \rightarrow abB \mid \epsilon$$

→ is given,

⇒ A string w generated by the given grammar- $w = ab$

Now, draw the parse trees



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

⌘ Problem 3:

⇒ Check whether the given grammar is ambiguous or not for the following string **aab**

$$S \rightarrow AB \mid aaB$$

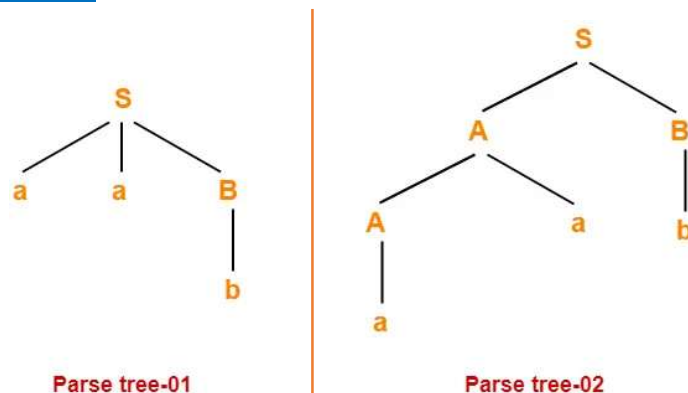
$$A \rightarrow a \mid Aa$$

$$B \rightarrow b$$

→ is given,

⇒ A string w generated by the given grammar- $w = aab$

Now, draw the parse trees



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

Problem 4:

⇒ Check whether the given grammar is ambiguous or not for the following string **aabbccdd**

$$S \rightarrow AB \mid C$$

$$A \rightarrow aAb \mid ab$$

$$B \rightarrow cBd \mid cd$$

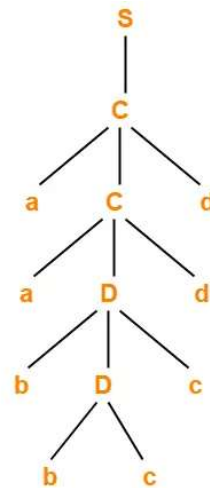
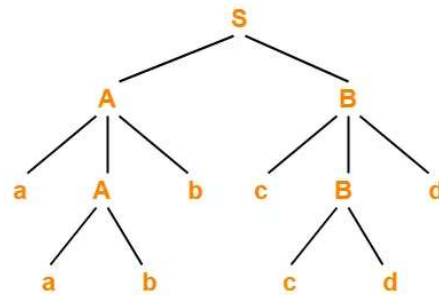
$$C \rightarrow aCd \mid aDd$$

$$D \rightarrow bDc \mid bc$$

→ is given,

⇒ A string w generated by the given grammar- $w = aabbccdd$

Now, draw the parse trees



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

Problem 5:

⇒ Check whether the given grammar is ambiguous or not for the following string **abababb**

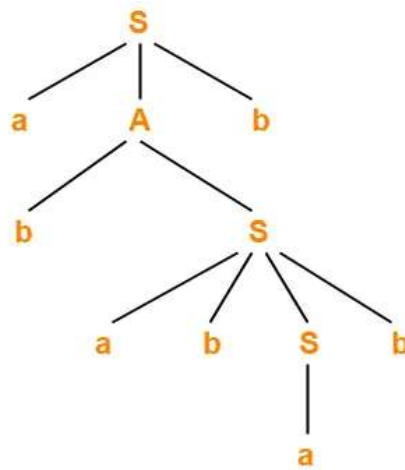
$$S \rightarrow a \mid abSb \mid aAb$$

$$A \rightarrow bS \mid aAb$$

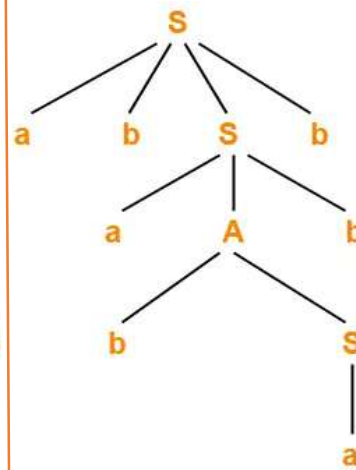
→ is given,

⇒ A string w generated by the given grammar- $w = abababb$

Now, draw the parse trees



Parse tree-01



Parse tree-02

Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

Problem 6:

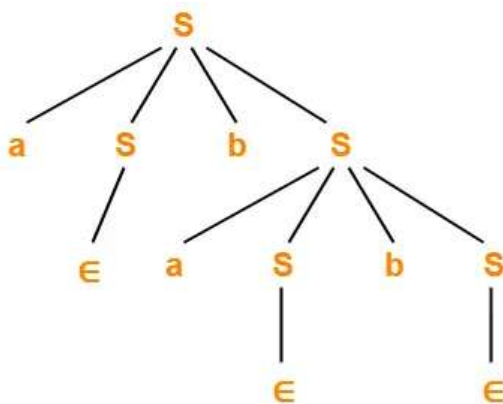
⇒ Check whether the given grammar is ambiguous or not for the following string **abab**

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

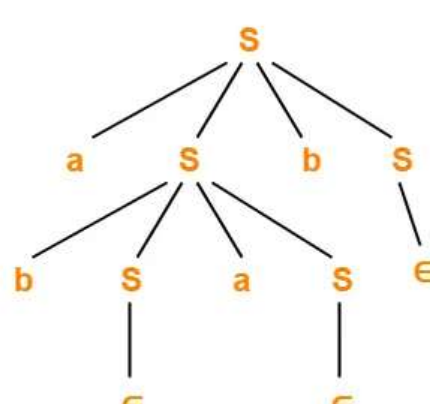
→ is given,

⇒ A string w generated by the given grammar- $w = abab$

Now, draw the parse trees



Parse tree-01



Parse tree-02

Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

❧ Problem 7:

⇒ Check whether the given grammar is ambiguous or not for the following string **id+id×id**

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow \text{id} \end{aligned}$$

→ There exists no string belonging to the language of grammar which has more than one parse tree.
Since a unique parse tree exists for all the strings, therefore the given grammar is unambiguous.

❧ Problem 8:

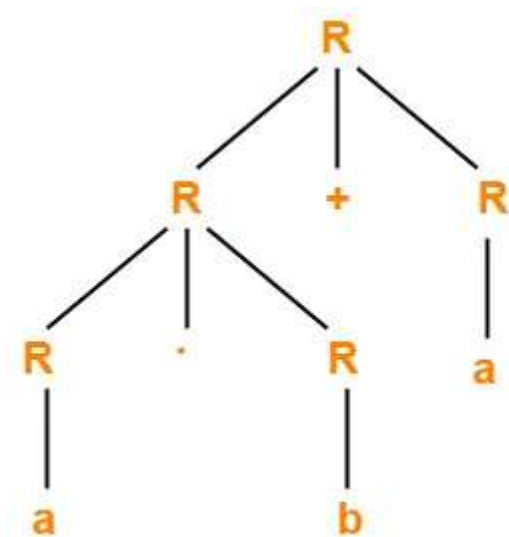
⇒ Check whether the given grammar is ambiguous or not-

$$R \rightarrow R + R \mid R \cdot R \mid R^* \mid a \mid b$$

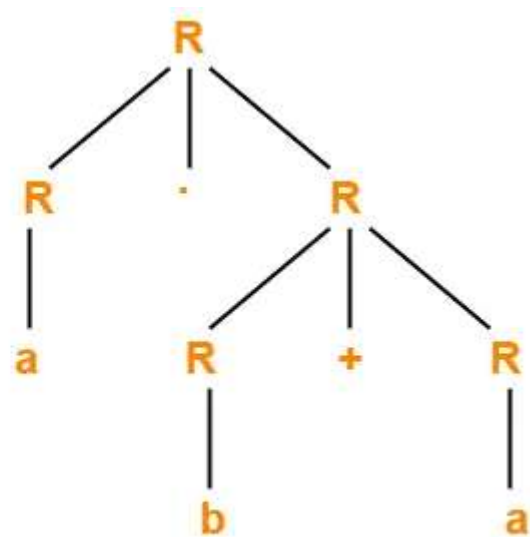
→ Let us,

⇒ consider a string w generated by the given grammar- $w = ab + a$

Now, draw the parse trees



Parse tree-01



Parse tree-02

Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

ASSOCIATIVITY & PRECEDENCE OF OPERATORS

Associativity of operators

⇒ **Associativity** tells us about the **order** in which operators of the **same precedence** are evaluated.

There are two main types:

1. **Left-Associative**
2. **Right Associative**

Left-Associative

→ Operators are evaluated **from left to right**.

Common left-associative operators:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)

→ **Example:**

$10 - 5 - 2 \rightarrow (10 - 5) - 2 \rightarrow 5 - 2 \rightarrow 3$

Right Associative

→ Operators are evaluated **from right to left**.

Common right-associative operators:

- =, +=, -=, etc. (assignment operators)
- ** (exponentiation in some languages like Python)

→ **Example:**

$a = b = c \rightarrow a = (b = c)$

Precedence of operators

⇒ **Precedence** determines **which operator** is evaluated **first** when an expression has **multiple different operators**.

Higher Precedence = Evaluated First

⇒ Operators with **higher precedence** are done **before** those with **lower precedence** unless **parentheses** are used.

Common Precedence Order (High → Low):

Precedence	Operator(s)	Description
1	()	Parentheses
2	\wedge or $**$	Exponent (in some langs)
3	$*$, $/$, $\%$	Multiplication, Division, Modulus
4	$+$, $-$	Addition, Subtraction
5	$=$, $+=$, $-=$	Assignment

Expression: $9 + 5 * 2$

There are two ways to read this:

- $(9 + 5) * 2 = 28$
- $9 + (5 * 2) = 19$

So, which one is correct?

Precedence Rule

- $*$ has higher precedence than $+$
- That means **multiplication happens first**

So, 5 is taken by $*$, not $+$

Therefore:

$$9 + (5 * 2) = 9 + 10 = 19$$

Note:

- **Associativity** helps when **same operators** are used (e.g., $9 - 5 - 2$)
- **Precedence** helps when **different operators** are used (e.g., $+$ and $*$)

SYNTAX DIRECTED TRANSLATION

Syntax Directed Translation

⇒ **Syntax-directed translation** means we add **rules or code** to grammar **productions** to describe how to **translate** or **process** the input.

Example:

Production:	$\text{expr} \rightarrow \text{expr} + \text{term}$
This means <code>expr</code> is the sum of two parts: another <code>expr</code> and a <code>term</code> .	

To translate this, we follow the structure using pseudo-code:

<pre>translate expr; // translate the left side translate term; // then the right side handle + ; // then handle the '+' operation</pre>
Each part is translated in order, based on how it appears in the production.

There are two notations for attaching semantic rules:

1. Syntax Directed Definitions
2. Syntax Directed Translation Schemes

Syntax Directed Definitions

⇒ **Syntax-Directed Definitions** are like an advanced version of context-free grammars.

They include:

1. Attributes

» Each grammar symbol has **attributes** (like variables holding values).

Example: **$\text{expr.t} \rightarrow \text{t}$** is an attribute of **`expr`**.

2. Semantic Rules

» Each production has **rules** to **calculate** or **assign** values to these attributes.

3. Annotated Parse Tree

- The parse tree becomes an **annotated parse tree**.
- Each node holds values for its attributes.
- These values are filled using the semantic rules.

Example:

Production rule:	$\text{expr} \rightarrow \text{expr} + \text{term}$
Add Attributes:	$\text{expr.t} = \text{expr1.t} + \text{term.t}$
This means the t value of the resulting expr is the sum of the t values of its children.	

Semantic Rule for a Grammar Production

Production rule:	$\text{expr} \rightarrow \text{expr} + \text{term}$
Semantic Rule:	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '+'$
Note: <ul style="list-style-type: none">\parallel is used for string concatenation..t is the attribute that stores the postfix string.This happens during syntax-directed translation.	

What It Means:

- This rule builds a **postfix expression**.
- The left side of $+$ is expr , the right side is term .
- expr.t holds the **postfix form** of the full expression.

So, the semantic rule says: Combine expr.t and term.t , then add $+$ at the end.

Postfix Notation:

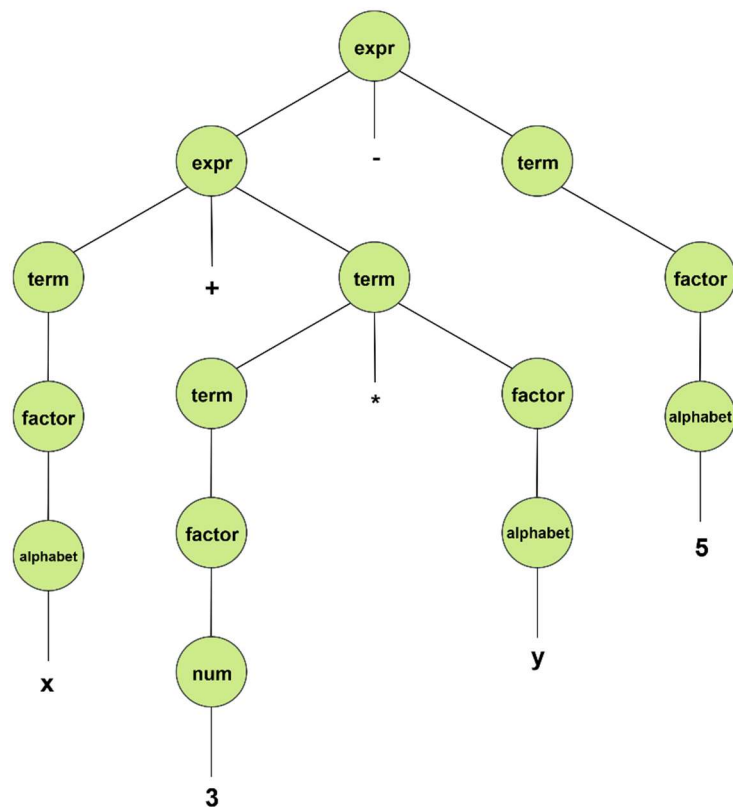
Example1:	Example2:
$\underbrace{a + b}_{\text{infix}} \rightarrow \underbrace{a b +}_{\text{postfix}}$	$\gg 9 - 5 + 2 \rightarrow \underbrace{95 -}_x + \underbrace{2}_y$ $\gg x + y \rightarrow xy +$ $\therefore 9 - 5 + 2 \rightarrow 95 - 2 +$

Grammar Of Production Rule

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$
$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$
$\text{factor} \rightarrow \text{num} \mid \text{alphabet} \mid (\text{expr})$
$\text{num} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$
$\text{alphabet} \rightarrow A \mid B \mid C \mid \dots \mid Z \mid a \mid b \mid c \mid \dots \mid z$

Example:

>>Show the parse tree for the following expressions $x + 3 * y - 5$



Productions and Semantic Rules for Postfix Conversion

Production	Semantic Rule
$\text{expr} \rightarrow \text{expr} + \text{term}$	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '+'$
$\text{expr} \rightarrow \text{expr} - \text{term}$	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '-'$
$\text{expr} \rightarrow \text{term}$	$\text{expr.t} := \text{term.t}$
$\text{term} \rightarrow 0$	$\text{term.t} := '0'$
$\text{term} \rightarrow 1$	$\text{term.t} := '1'$
$\text{term} \rightarrow 2$	$\text{term.t} := '2'$
...	...
$\text{term} \rightarrow 9$	$\text{term.t} := '9'$

>> We applied semantic rules for each production of a context-free grammar. Now we will see how semantic rules are embedded in parse tree.

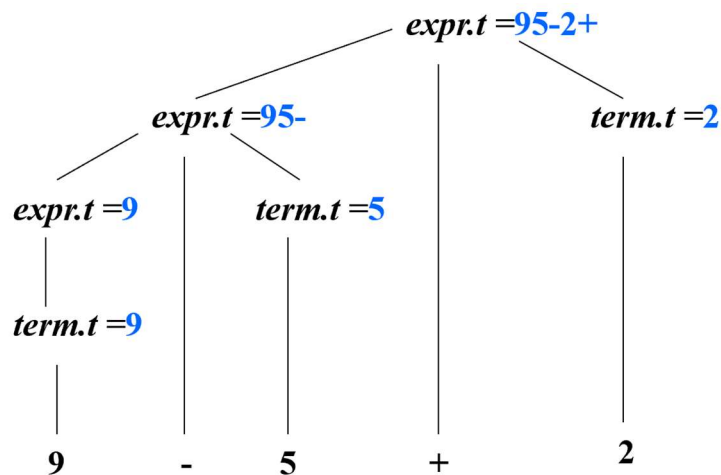


Fig: Annotated Parse Tree

What is an Annotated Parse Tree?

- It's a **regular parse tree** with **extra information**.
- Each **node** in the tree represents a **grammar symbol**.
- Each node also has **attributes** (like .t) whose values are **computed using semantic rules**.

Attribute Grammars

⇒ In an **attribute grammar**, every **grammar symbol** (like `expr`, `term`, etc.) has **attributes** that store extra information.

These attributes can represent things like:

- The **value** of an expression (e.g., $1 + 2 = 3$)
- The **data type** (e.g., `int`, `float`)
- The **memory location** of a variable
- The **translated code** for statements or expressions

Types of Attributes

1. Synthesized Attributes

- Computed **from children** nodes (bottom-up)
- Flow **upward** in the parse tree
- Common in **expression evaluation** and **code generation**

Example:

For $\text{expr} \rightarrow \text{expr} + \text{term}$,
 $\text{expr.value} = \text{expr.value} + \text{term.value}$

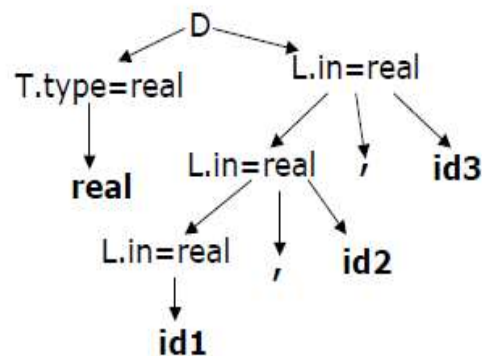
2. Inherited Attributes

- Passed **from parent or siblings** (top-down or sideways)
- Flow **downward or sideways** in the tree
- Useful for things like **type checking**, **scope info**, etc.

Example:

Passing **data type** from parent to child node for variable declarations

Production	Semantic rules
$D ::= T L$	$L.in := T.type$
$T ::= \text{int}$	$T.Type := \text{integer}$
$T ::= \text{real}$	$T.type := \text{real}$
$L ::= L1 , id$	$L1.in := L.in$ $Addtype(id.entry, L.in)$
$L ::= id$	$Addtype(id.entry, L.in)$



Syntax Directed Translation Schemes

⇒ A Translation Scheme is a Context-Free Grammar extended with:

1. Attributes

- Each grammar symbol can have **attributes** (like values, types, code, etc.)

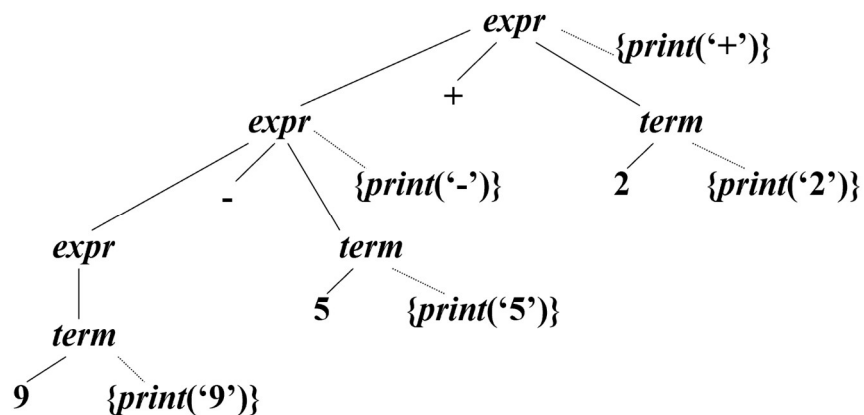
2. Semantic Actions

- Special **code fragments** written in **{ }** (braces)
- Inserted **inside the production rules**
- These actions are executed **during parsing**

Productions with Semantic Actions

Production	Semantic Action
$\text{expr} \rightarrow \text{expr} + \text{term}$	<code>{print('+')}</code>
$\text{expr} \rightarrow \text{expr} - \text{term}$	<code>{print('-')}</code>
$\text{expr} \rightarrow \text{term}$	<code>{print()} (no operator)</code>
$\text{term} \rightarrow 0$	<code>{print('0')}</code>
$\text{term} \rightarrow 1$	<code>{print('1')}</code>
...	...
$\text{term} \rightarrow 9$	<code>{print('9')}</code>

>> Translation Scheme = SDD + explicit order of semantic actions



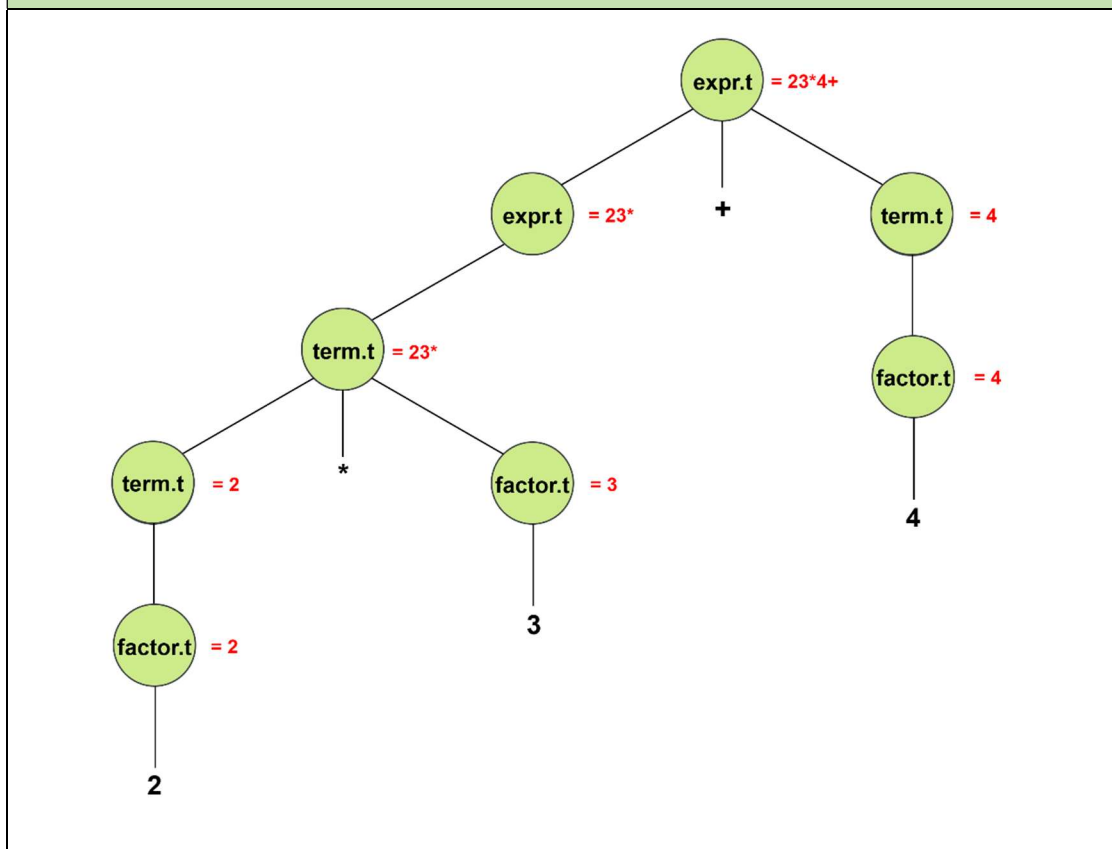
⌘ Problem 1:

>>Show the annotated parse tree for the following expressions $2 * 3 + 4$.

>>Grammar productions we use:

Production	Semantic Rule
$\text{expr} \rightarrow \text{expr} + \text{term}$	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '+'$
$\text{expr} \rightarrow \text{term}$	$\text{expr.t} := \text{term.t}$
$\text{term} \rightarrow \text{term} * \text{factor}$	$\text{term.t} := \text{term.t} \parallel \text{factor.t} \parallel '*'$
$\text{term} \rightarrow \text{factor}$	$\text{term.t} := \text{factor.t}$
$\text{factor} \rightarrow 0$	$\text{factor.t} := '0'$
$\text{factor} \rightarrow 1$	$\text{factor.t} := '1'$
$\text{factor} \rightarrow 2$	$\text{factor.t} := '2'$
...	...
$\text{factor} \rightarrow 9$	$\text{factor.t} := '9'$

Annotated Parse Tree:



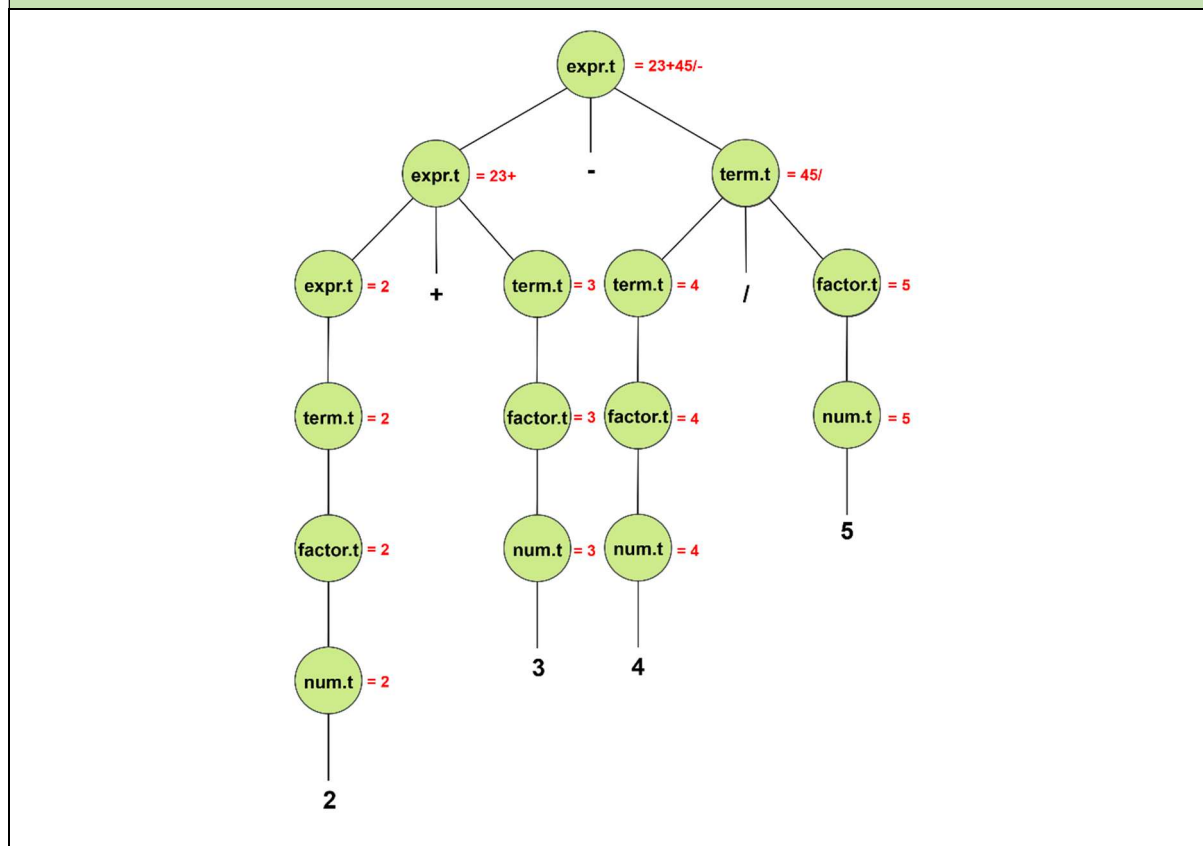
⌘ Problem 2:

>>Show the annotated parse tree for the following expressions $2 + 3 - 4/5$.

>>Grammar productions we use:

Production	Semantic Rule
$\text{expr} \rightarrow \text{expr} + \text{term}$	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '+'$
$\text{expr} \rightarrow \text{expr} - \text{term}$	$\text{expr.t} := \text{expr.t} \parallel \text{term.t} \parallel '-'$
$\text{expr} \rightarrow \text{term}$	$\text{expr.t} := \text{term.t}$
$\text{term} \rightarrow \text{term} / \text{factor}$	$\text{term.t} := \text{term.t} \parallel \text{factor.t} \parallel '/'$
$\text{term} \rightarrow \text{factor}$	$\text{term.t} := \text{factor.t}$
$\text{factor} \rightarrow \text{num}$	$\text{factor.t} := \text{num.t}$
$\text{num} \rightarrow 0$	$\text{num.t} := '0'$
$\text{num} \rightarrow 1$	$\text{num.t} := '1'$
$\text{num} \rightarrow 2$	$\text{num.t} := '2'$
...	...
$\text{num} \rightarrow 9$	$\text{num.t} := '9'$

Annotated Parse Tree:



✂ **Problem3:** What is the annotated parse tree? Explain the term Synthesized attribute with an example.

➤ Annotated Parse Tree:

⇒ An **annotated parse tree** is a **syntax tree** where **attributes are attached to each node**, representing **semantic information** (like data types, values, or symbol table entries) computed during parsing.

- It is built during **syntax-directed translation**.
- Helps in **semantic analysis** and **intermediate code generation**.

➤ Synthesized Attribute:

- A **synthesized attribute** is an attribute whose value is **calculated from its children** in the parse tree (i.e., passed **bottom-up**).
- Used in **bottom-up parsing**.

Example:

For the rule:

`expr → expr + term { expr.t := expr.t || term.t || '+' }`

`expr → term { expr.t := term.t }`

`term → 2 { term.t := '2' }`

`term → 3 { term.t := '3' }`

Problem 4:

>>Construct a Context Free Grammar for arithmetic expression of digits (0 to 9) with the four binary operators (+, -, *, /) keeping the associativity and precedence of the arithmetic binary operators. Show that your grammar is correct by constructing a Parse Tree for this expression:

" $id = 5 * 4 - 8/2$ "

Solutions

Context Free Grammar:

$S \rightarrow id = E$

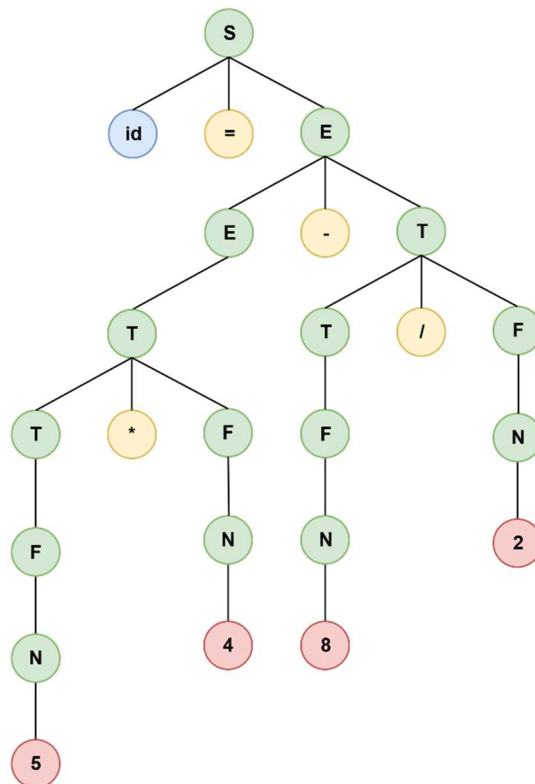
$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T/F \mid T * F \mid F$

$F \rightarrow N \mid (E)$

$N \rightarrow 0 \mid 1 \dots \mid 9$

Parse Tree:



Problem 5:

>>Construct a Context Free Grammar for arithmetic expression of digits (0 to 9) with the four binary operators (+, -, *, /) keeping the associativity and precedence of the arithmetic binary operators. Show that your grammar is correct by constructing a Parse Tree for this expression:

" $id = (x + 13) * y - 5$ "

Solutions

Context Free Grammar:

$S \rightarrow id = E$

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T / F \mid T * F \mid F$

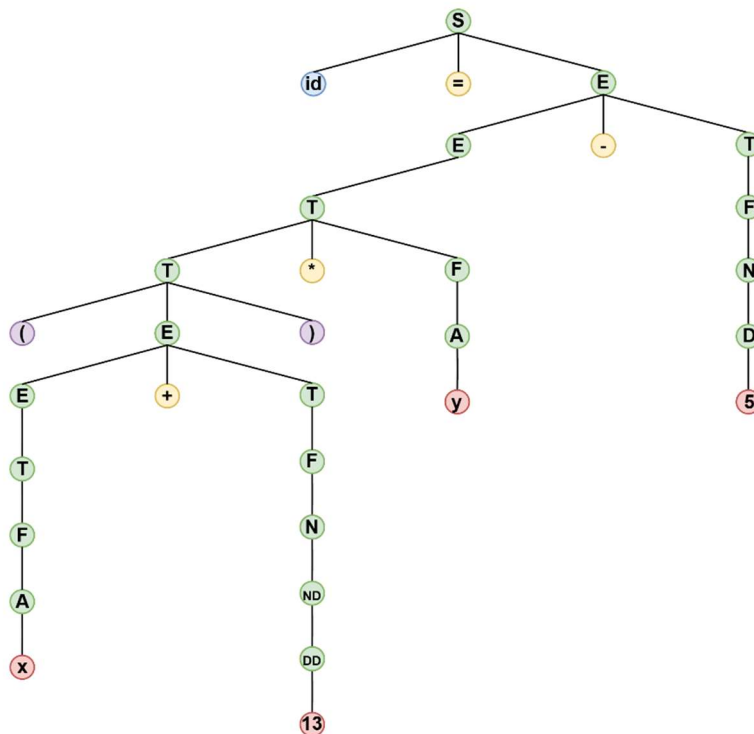
$F \rightarrow N \mid A \mid (E)$

$N \rightarrow ND \mid D$

$D \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

$A \rightarrow a \mid b \mid c \mid \dots \mid z$

Parse Tree:



Problem 5:

Using the following context free grammar, construct a syntax-directed translations scheme that translates arithmetic expression from infix notation into postfix notation. Show the application of your scheme to the string “a^b+a%b”.

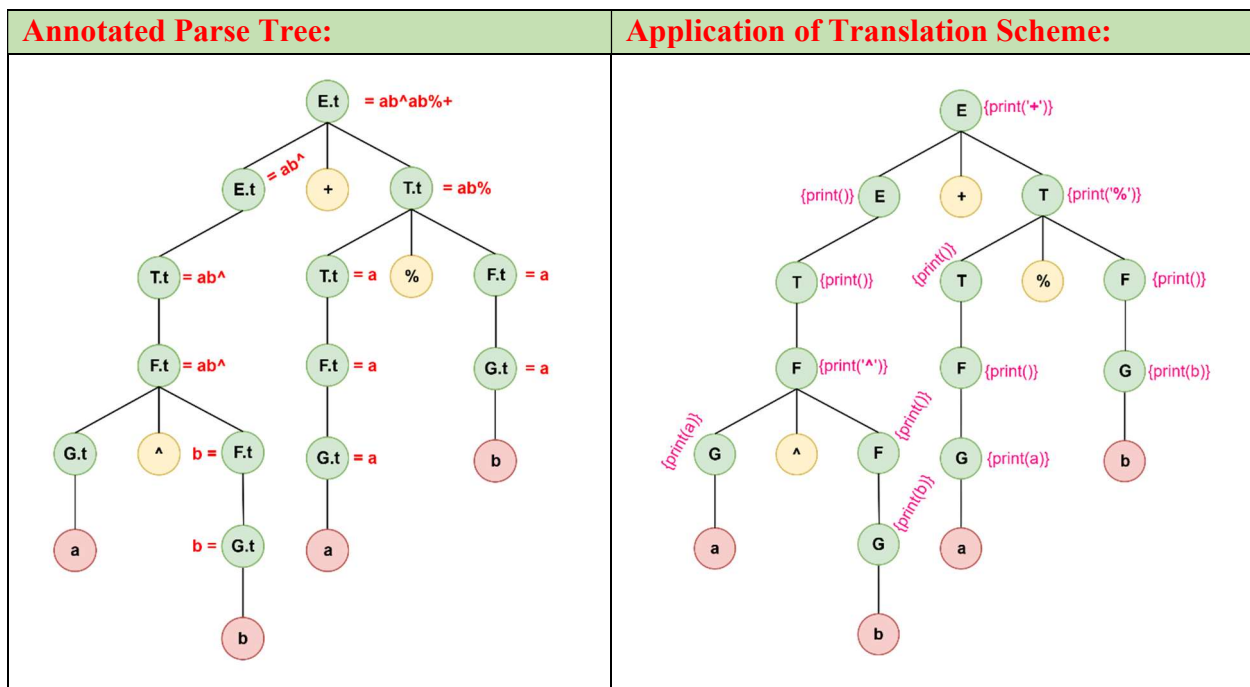
$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \% F \mid F$$

$$F \rightarrow G^{\wedge} F \mid G$$

$$G \rightarrow a \mid b$$

Production	Semantic Rule	Semantic Scheme
$E \rightarrow E + T$	$E.t := E.t \parallel T.t \parallel '+'$	$\{\text{print}(' + ')\}$
$E \rightarrow T$	$E.t := T.t$	$\{\text{print}()\}$
$T \rightarrow T \% F$	$T.t := T.t \parallel F.t \parallel '\%'$	$\{\text{print}(' \% ')\}$
$T \rightarrow F$	$T.t := F.t$	$\{\text{print}()\}$
$F \rightarrow G^{\wedge} F$	$F.t := G.t \parallel F.t \parallel '^'$	$\{\text{print}(' ^ ')\}$
$F \rightarrow G$	$F.t := G.t$	$\{\text{print}()\}$
$G \rightarrow a$	$G.t := 'a'$	$\{\text{print}(' a ')\}$
$G \rightarrow b$	$G.t := 'b'$	$\{\text{print}(' b ')\}$



Practice Problem

✂ **Problem1:** For the following statement show output of different phases of a Compiler, where A, C, Z, E are the float type variables.

$$Z = 20 - (A * C / E - 5.67) + A$$

⇒ **Solution:**

Step1: Lexical Analyzer	Step2: Syntax Analyzer
<p> $Z = 20 - (A * C / E - 5.67) + A$ $\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$ $\langle id,1 \rangle \langle = \rangle \langle 20 \rangle \langle - \rangle \langle (\rangle \langle id,2 \rangle \langle * \rangle \langle id,3 \rangle \langle / \rangle \langle id,4 \rangle \langle - \rangle \langle 5.67 \rangle \langle) \rangle \langle + \rangle \langle id,2 \rangle$ </p> <p>Final Output: $id1 = 20 - (id2 * id3 / id4 - 5.67) + id2$</p>	
Step3: Semantic Analyzer	Step4: Intermediate Code Generator
	<pre> t1 = id2 * id3 t2 = t1/id4 t3 = t2 - 5.62 t4 = inttofloat(20) t5 = t4 - t3 t6 = t5 + id2 id1 = t6 </pre>
Step5: Code Optimizer	Step6: Code Generator
<pre> t1 = id2 + id3 t2 = t1/id4 t3 = t2 - 5.67 t4 = t3 - 20.0 id1 = t4 + id2 </pre>	<pre> MOVF R2, id2 MOVF R3, id3 ADDF R3, R2 MOVF R4, id4 DIVF R3, R4 SUBF R3, #5.67 LDIF R7, 20 SUBF R7, R3 ADDF R7, R2 MOVF id1, R7 </pre>

Problem2:

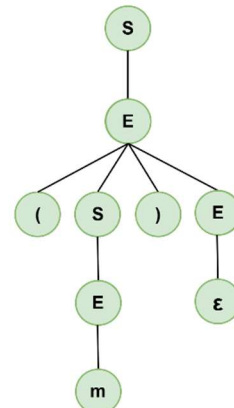
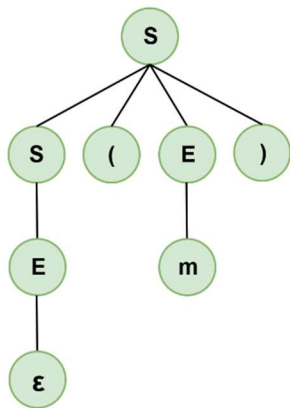
>> Create a string to prove that the following grammar is ambiguous using parse tree.

$$S \rightarrow S(E) \mid E$$

$$E \rightarrow (S)E \mid m \mid n \mid \varepsilon$$

>> Consider a string w generated by the given grammar- $w = "(m)"$

Now, draw the parse trees:



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

Problem3:

>> Consider the following **context free grammar**. Show how the string “**ceaedbe**” can be generated by this grammar using **left derivation**.

$$S \rightarrow SaA \mid A$$

$$A \rightarrow AbB \mid B$$

$$B \rightarrow cSd \mid e$$

>> Let's Generated the string “**ceaedbe**”:

$$S \rightarrow A$$

$$S \rightarrow \mathbf{A}bB$$

$$S \rightarrow \mathbf{cS}dbB$$

$$S \rightarrow \mathbf{cSa}dbB$$

$$S \rightarrow \mathbf{cAa}dbB$$

$$S \rightarrow \mathbf{cBa}dbB$$

$$S \rightarrow \mathbf{cea}dbB$$

$$S \rightarrow \mathbf{ceaB}dbB$$

$$S \rightarrow \mathbf{ceaB}dbB$$

$$S \rightarrow \mathbf{cea}db\mathbf{B}$$

$$S \rightarrow \mathbf{cea}db\mathbf{e}$$

Problem 4:

»Construct a Context Free Grammar for arithmetic expression of digits (0 to 9) with the four binary operators (+, -, *, /) keeping the associativity and precedence of the arithmetic binary operators. Show that your grammar is correct by constructing a **Parse Tree** for this expression:

"9 * 2 - 6 / 3"

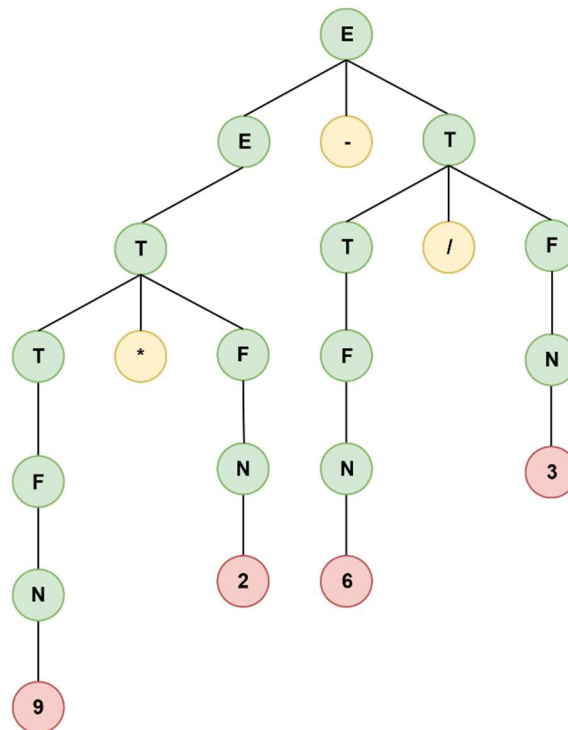
Solutions

Context Free Grammar:

$$E \rightarrow E + T \mid E - T \mid T$$
$$\mathbf{T} \Rightarrow \mathbf{T}/\mathbf{F} \mid \mathbf{T} * \mathbf{F} \mid \mathbf{F}$$
$$\mathbf{F} \rightarrow \mathbf{N}$$

N \rightarrow **0** | **1 ...** | **9**

Parse Tree:



Problem 5:

>>Consider the following production rules.

$B \rightarrow B \text{ NAND } C \mid C$

$C \rightarrow C \text{ NOR } D \mid D$

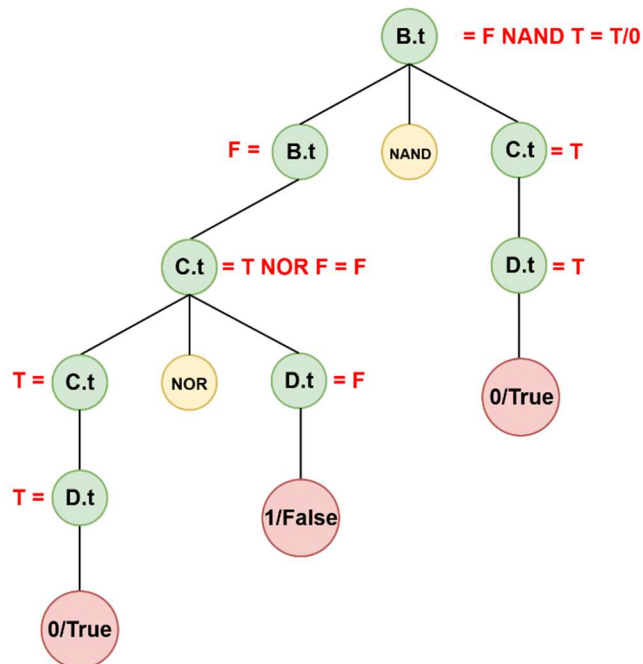
$D \rightarrow \text{not } D \mid (B) \mid 0 \mid 1$

The NAND and NOR serve as logical operators that work as the opposites of AND and OR, respectively. Write the semantic rules to compute the truth value of the Boolean expression: "(0 NOR 1) NAND 0", where 0 and 1 represent constants (with 0 as True and 1 as False), and show the annotated parse tree

>>Grammar productions we use:

Production	Semantic Rule
$B \rightarrow B \text{ NAND } C$	$B.t := B.t \parallel C.t \parallel \text{'NAND'}$
$B \rightarrow C$	$B.t := C.t$
$C \rightarrow C \text{ NOR } D$	$C.t := C.t \parallel D.t \parallel \text{'NOR'}$
$C \rightarrow D$	$C.t := D.t$
$D \rightarrow 0$	$D.t := \text{'0'}/\text{'True'}$
$D \rightarrow 1$	$D.t := \text{'1'}/\text{'False'}$

Annotated Parse Tree:



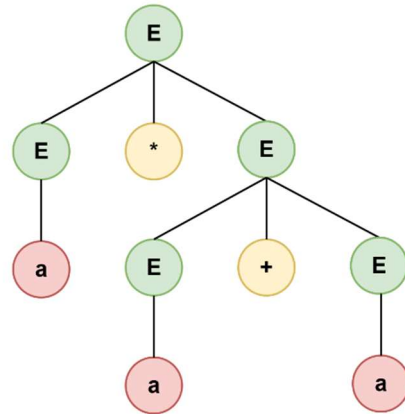
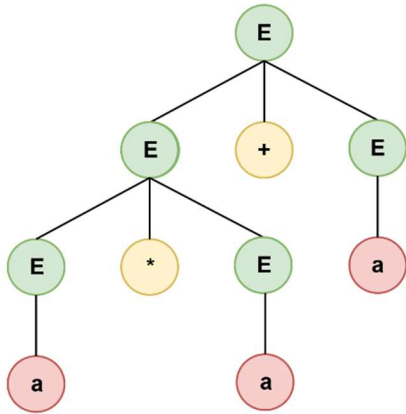
Problem6:

>> Prove that the following grammar is ambiguous

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

>> Consider a string w generated by the given grammar- w = "a*a+b"

Now, draw the parse trees:



Since two different parse trees exist for string w, therefore the given grammar is ambiguous.

Problem7:

>> Add "Semantic Rules" and "Translation Schemes" to the following productions of a context free grammar.

$$S \rightarrow U$$

$$U \rightarrow T a U \mid T a T$$

$$T \rightarrow a T b T \mid b T a T \mid d$$

Production	Semantic Rule	Semantic Scheme
$S \rightarrow U$	$S.t := U.t$	$\{\text{print}()\}$
$U \rightarrow T a U$	$U.t := T.t \parallel U.t \parallel 'a'$	$\{\text{print}('a')\}$
$U \rightarrow T a T$	$U.t := T.t \parallel U.t \parallel 'a'$	$\{\text{print}('a')\}$
$T \rightarrow a T b T$	$T.t := T.t \parallel T.t \parallel 'a' \parallel 'b'$	$\{\text{print}('a', 'b')\}$
$T \rightarrow b T a T$	$T.t := T.t \parallel T.t \parallel 'b' \parallel 'a'$	$\{\text{print}('b', 'a')\}$
$T \rightarrow d$	$T.t := 'd'$	$\{\text{print}('d')\}$

Problem8:

>> Create a string to prove that the following grammar is ambiguous using parse tree.

$S \rightarrow XY \mid W$

$X \rightarrow @X\$ \mid \epsilon$

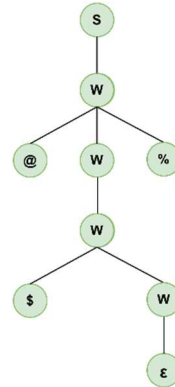
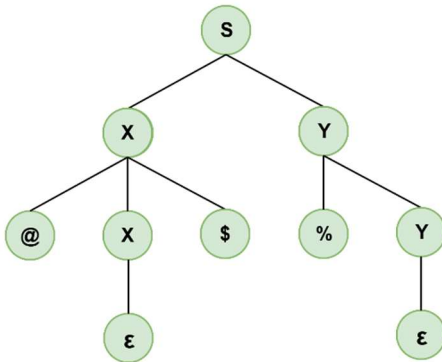
$Y \rightarrow \%Y \mid \epsilon$

$W \rightarrow @W\% \mid Z$

$Z \rightarrow \$Z \mid \epsilon$

>> Consider a string w generated by the given grammar- $w = "@\$%"$

Now, draw the parse trees:



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.

Problem9:

>> Create a string to prove that the following grammar is ambiguous using parse tree.

$S \rightarrow AB \mid BC$

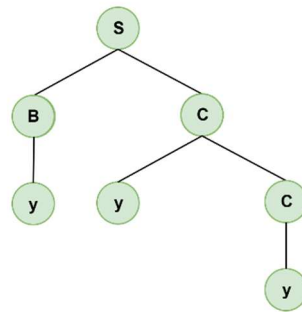
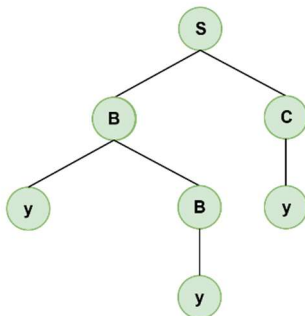
$A \rightarrow xA \mid x$

$B \rightarrow yB \mid y$

$C \rightarrow yC \mid y$

>> Consider a string w generated by the given grammar- $w = "yyy"$

Now, draw the parse trees:



Since two different parse trees exist for string w , therefore the given grammar is ambiguous.