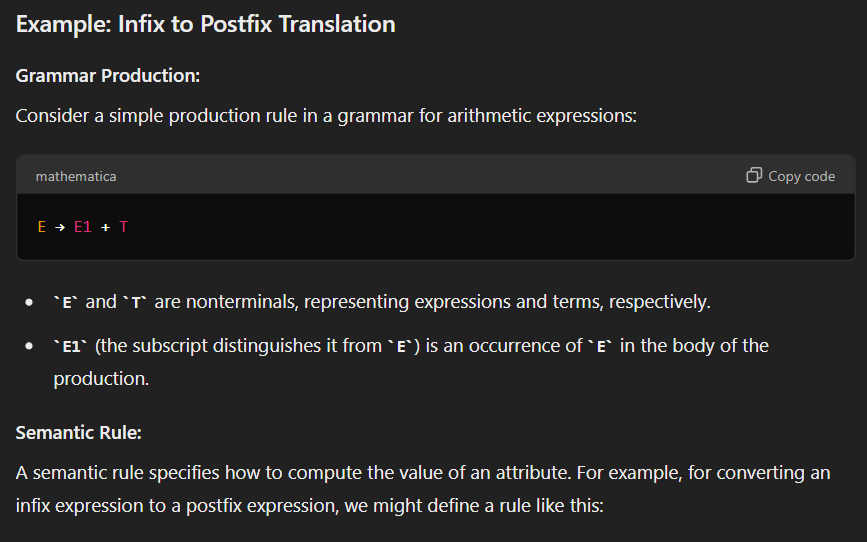
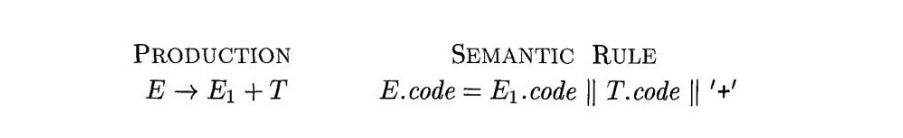
# **Syntax-Directed Translation (SDT)**

A Syntax-Directed Translation (SDT) is a method used in compilers to specify how the values of attributes are computed.

These attributes can be used for various purposes, such as translating expressions or performing type checking.

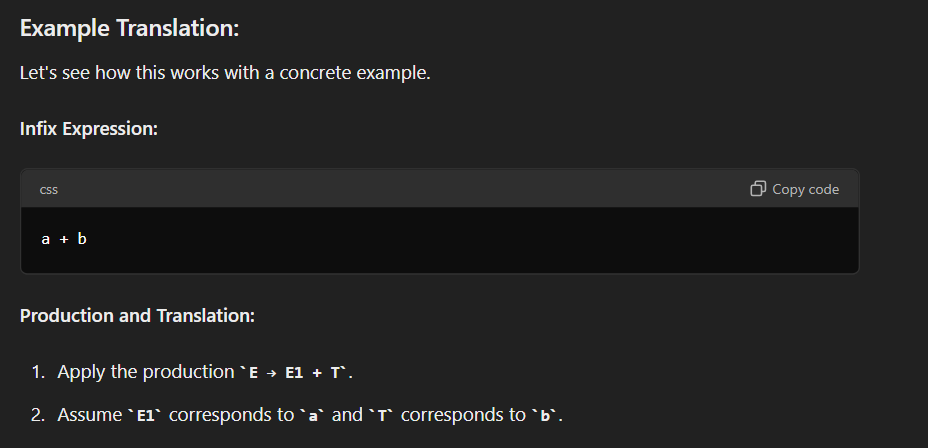
In SDT, we associate semantic rules with the grammar productions.

****

****

Both E and T have a string-valued attribute code. The semantic rule specifies that the string E.code is formed by concatenating E.code, T.code, and the character '+’.

**Example:**

****

****

A **syntax-directed definition (SDD)** is a context-free grammar together with, attributes and rules.

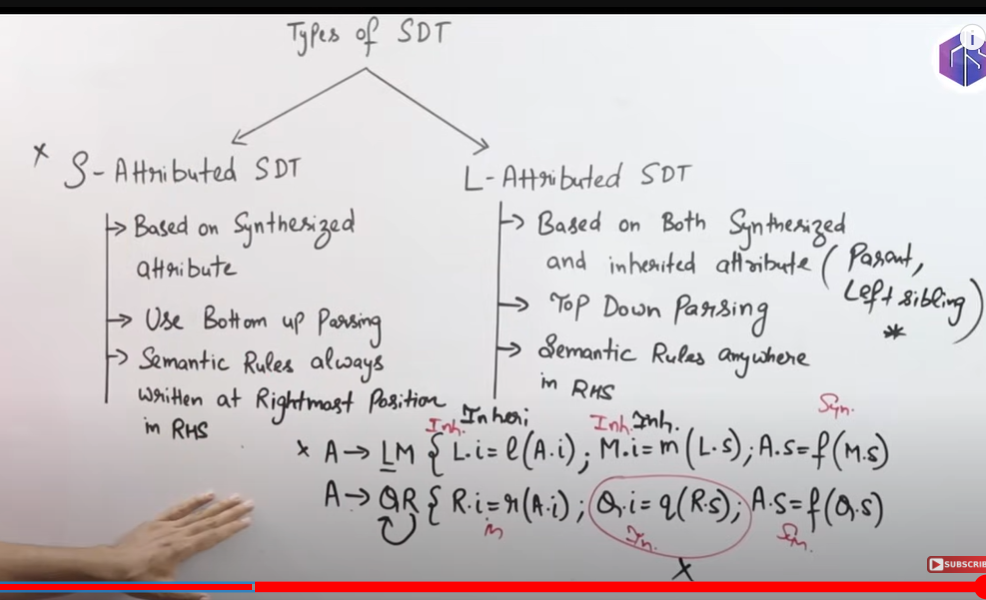
A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**

**Attributes:**

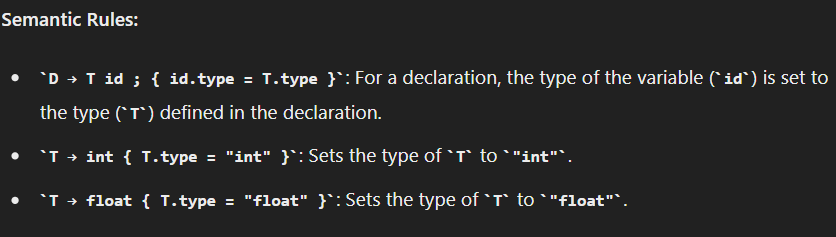
1. Synthesized attributes
2. Inherited attributes

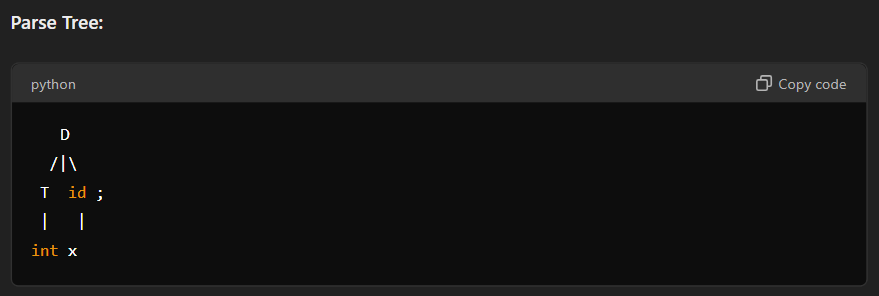
**SDT:**

1. S-Attributed SDT
2. L-Attributed SDT

****

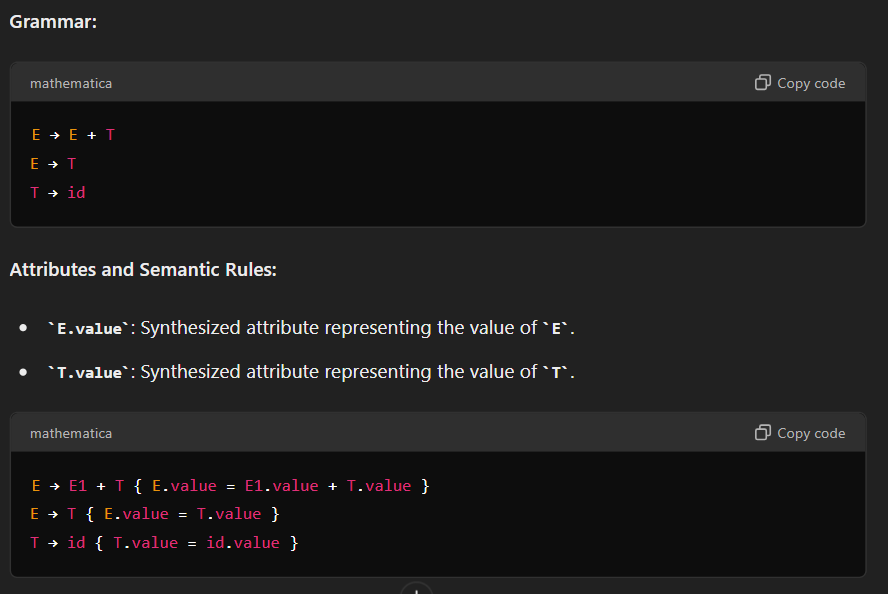
## **LA SDT**

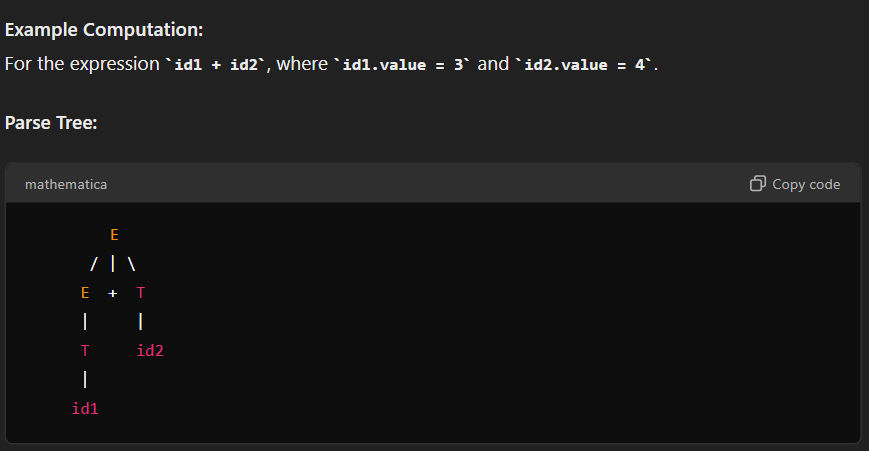


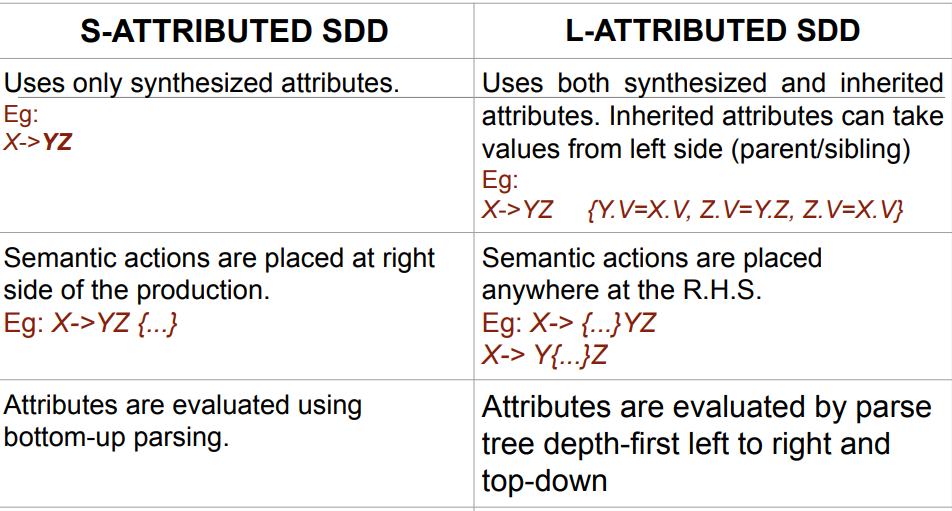


# 

## **S-Attributed SDT**





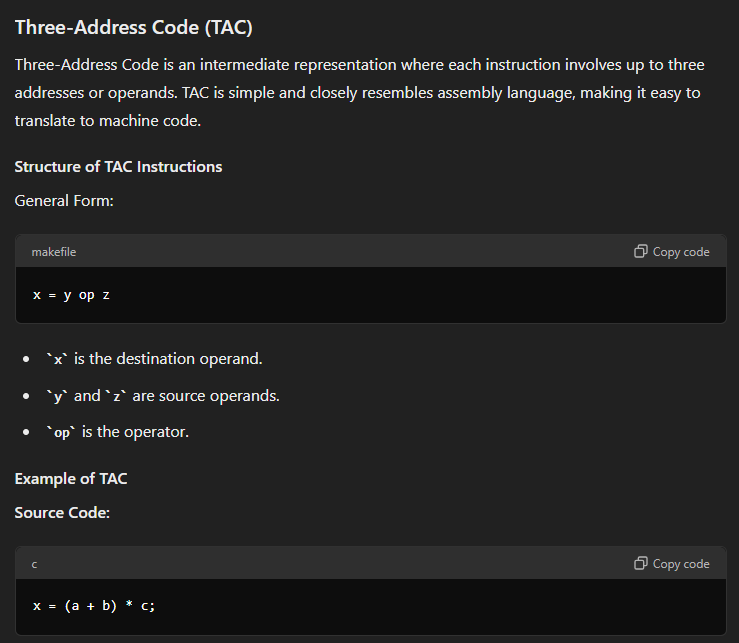


# **Intermediate Code Generation**

**Intermediate Code Generation** is a key phase in the compiler design process, which involves translating high-level source code into an intermediate representation. This intermediate code sits between the high-level source code and the low-level machine code or assembly code. The purpose of intermediate code generation is to simplify the translation process and make it easier to perform various optimizations and analyses.

## **Three-Address Code (TAC)**

**Three-Address Code (TAC)** is a type of intermediate representation used in compilers. It provides a way to express operations with up to three addresses or operands in each instruction. TAC is designed to be simple and close to machine code, making it useful for various compiler optimizations and translations.



# 

# 

# 

# 

# 

# 

# 

# **Error recovery strategies:**

In compiler design, error recovery strategies are techniques used by the compiler to handle and recover from errors encountered during the compilation process. These strategies ensure that the compilation process can continue and provide meaningful feedback to the programmer, even in the presence of syntax or semantic errors.

1. **Panic Mode**

Panic mode recovery is one of the **simplest and most effective** error recovery strategies. When a syntax error is detected, the parser discards input symbols one at a time until it finds a designated set of synchronizing tokens, such as statement terminators (e.g., semicolons) or keywords (e.g., if, while).

This approach prevents the parser from getting stuck in a loop and ensures that it can continue processing the rest of the input. The simplicity of this method makes it **easy to implement**, and it is **widely used in practice.**

**Practical Scenario:**

Consider a situation where the parser encounters an unexpected token in the middle of a statement. For example, in the code snippet:

int main() {

int x = 10;

x = x + 5;

y = x \* 2; // Error: 'y' is undeclared

return 0;

}

Upon encountering the undeclared variable y, the parser enters panic mode, skips the erroneous part of the statement, and resumes parsing after the semicolon. This allows the parser to continue processing the subsequent code, diagnosing other potential errors.

1. **Phrase-Level Recovery**

Phrase-level recovery involves local correction of the input. When an error is detected, the parser makes minimal changes to the input, such as inserting, deleting, or replacing tokens, to allow the parser to continue. This method attempts to correct the error on the spot and proceed with parsing.

Phrase-level recovery is more sophisticated than panic mode. The corrections should be minimal to ensure that the parser can continue with as few disruptions as possible. This method is beneficial because it often results in fewer discarded tokens, preserving more of the original input.

**Practical Scenario:**

Imagine a parser encountering a missing semicolon in the code:

int main() {

int x = 10

x = x + 5;

return 0;

}

When the parser detects the missing semicolon after int x = 10, it can insert the semicolon, correcting the error:

int main() {

int x = 10;

x = x + 5;

return 0;

}

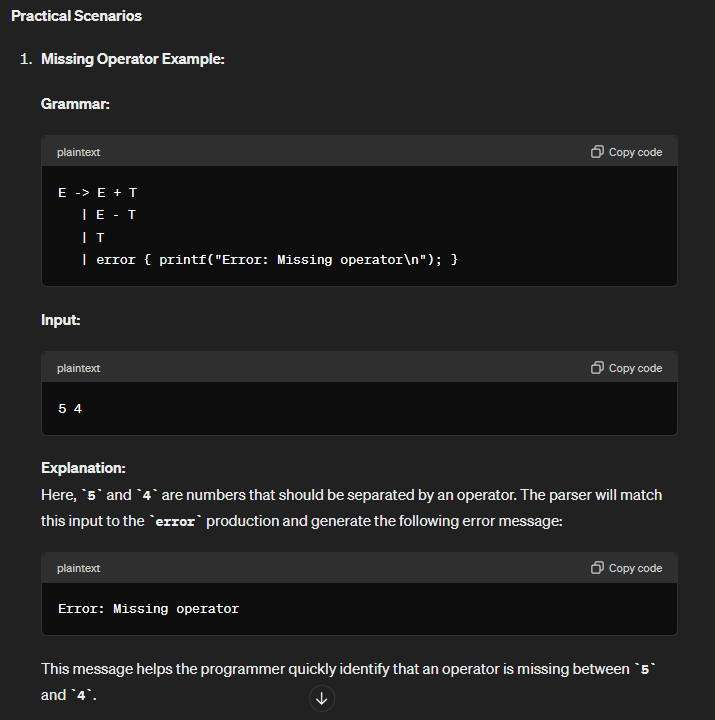
This allows the parser to continue parsing the rest of the code smoothly, resulting in a meaningful error recovery and fewer interruptions.

1. **Error Productions:**

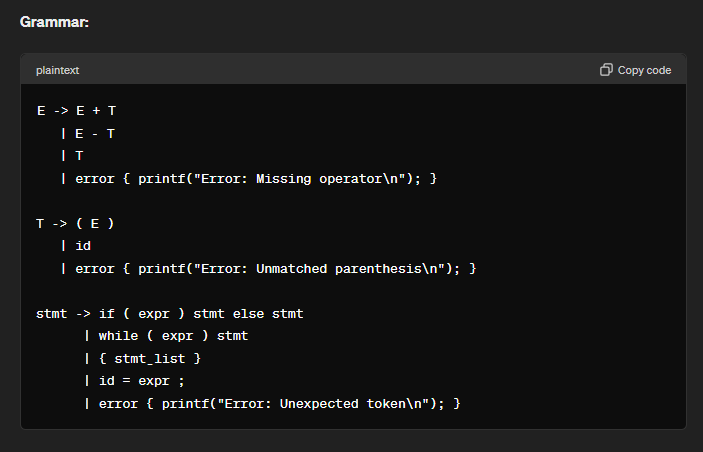
Error Productions are grammar rules added specifically to handle common syntax errors. These productions allow the parser to recognize and provide meaningful error messages for specific mistakes, enabling the parser to continue processing the input.

This approach integrates error handling directly into the grammar, making it easier to diagnose and correct mistakes.

While error productions improve error detection and recovery, they must be used judiciously to avoid making the grammar overly complex or ambiguous.



We can also write multiple error message.



1. **Global Correction:**

Global correction aims to find the smallest number of changes needed to transform the erroneous input into a valid one. This strategy involves analyzing the entire input and computing the minimal edit distance, which can be computationally expensive.

Global correction provides the **most accurate error recovery** by considering the entire input context. It often results in the least disruptive corrections and helps maintain the **integrity** of the input.

However, the **complexity** of this approach can be a drawback, as it may require significant computational resources.

**Practical Scenario:**

Imagine a parser using global correction to handle a series of nested errors in a complex program:

int main() {

int x = 10;

for (int i = 0 i < 10; i++) { // Error: missing semicolon

x = x + i

print(x); // Error: missing semicolon

}

return 0;

}

The global correction strategy would analyze the entire input and determine that inserting semicolons after 0, x + i, and print(x) would correct the errors with minimal changes. This allows the parser to recover gracefully and provide a corrected version of the code.

**Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest**.

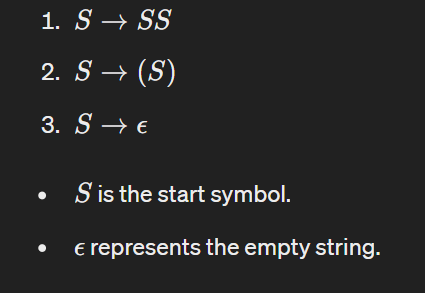
**Context-Free Grammars (CFGs)** are a type of formal grammar that are particularly useful for describing the syntax of programming languages. They can be classified into recursive and non-recursive grammars. Let’s define each type and provide examples to clarify the differences.

**Recursive CFGs**

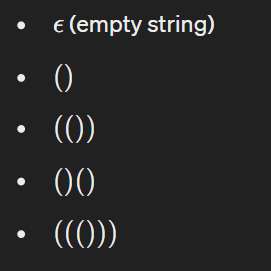
A recursive CFG contains at least one production rule where a non-terminal symbol is defined in terms of itself. This allows the grammar to generate infinitely long strings and is essential for capturing many syntactic structures in programming languages, like nested expressions or balanced parentheses.

**Example of a Recursive CFG:**

Consider a simple grammar for balanced parentheses:



This grammar can generate strings like:



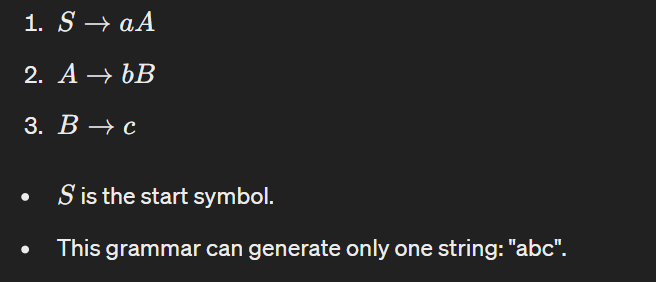
In this grammar, rule 1 is recursive because it defines 𝑆 in terms of itself.

**Non-Recursive CFGs**

A non-recursive CFG does not have any production rules where a non-terminal symbol is defined in terms of itself. This type of grammar can only generate a finite set of strings and is often used for simpler languages or specific constructs within a larger language.

**Example of a Non-Recursive CFG:**

Consider a grammar for a finite set of strings:



In this grammar, there are no recursive rules, so it can only generate the finite set of strings defined by the production rules.

**Comparison and Justification of Which is Better**

**Expressiveness:**

* **Recursive CFGs** are more powerful in terms of expressiveness. They can describe an infinite number of strings and can model complex syntactic structures such as nested or recursive patterns. This is essential for programming languages, where constructs like nested expressions, block structures, and recursion are common.
* **Non-Recursive CFGs** are limited to generating a finite number of strings, making them suitable for simpler, more restricted languages or subsets of languages.

**Parsing and Implementation:**

* **Recursive CFGs** might require more sophisticated parsing algorithms, such as recursive descent parsers or more advanced LR parsers, which can handle the recursive nature of the grammar. This complexity is necessary for the flexibility and expressiveness they provide.
* **Non-Recursive CFGs** are simpler to parse and implement. They can be processed by straightforward parsing techniques like table-driven parsers or simple finite automata.

**Which One is Better?**

The choice between recursive and non-recursive CFGs depends on the application:

* **For full-featured programming languages** and complex syntactic structures, recursive CFGs are better. Their ability to handle nested constructs and infinite patterns is crucial for accurately representing the syntax of such languages.
* **For specific, limited domains or simple languages,** non-recursive CFGs might be more appropriate. They are easier to implement and understand, and their limited scope can be beneficial in cases where the full power of recursive grammars is not needed.

**Conclusion**

While non-recursive CFGs have their place in certain applications due to their simplicity and ease of implementation, recursive CFGs are generally better for most practical purposes, especially in the context of programming languages, due to their greater expressiveness and ability to model complex, nested structures.

**Compare the Error Recovery strategies and state advantages and disadvantages of each of it. Which one is better in your opinion and why. Add necessary examples?**

**1. Panic Mode Recovery**

**How it works**: When the parser finds an error, it skips parts of the input until it finds a specific set of symbols that it knows can follow the error, called "synchronizing tokens" (like semicolons in many languages).

**Example**: If you have a missing semicolon in a line of code, the parser skips ahead to the next semicolon it finds.

**Advantages:**

* Simple and fast to implement.
* Quickly gets past the error, allowing the parser to continue checking the rest of the code.

**Disadvantages:**

* Can skip a lot of code, missing multiple errors.
* Not always precise in pinpointing the error location.

**2. Phrase-Level Recovery**

**How it works:** The parser makes small changes to the input, like inserting or deleting tokens, to try and continue parsing.

**Example**: If a semicolon is missing, the parser might insert one to see if it can continue.

**Advantages:**

* More precise in handling errors.
* Can often correct the error without skipping much code.

**Disadvantages:**

* More complex to implement than panic mode.
* Can sometimes lead to incorrect assumptions about the nature of the error.

**3. Error Productions**

**How it works:** The grammar itself includes specific rules for common errors. When the parser encounters these errors, it uses these special rules to handle them.

**Example:** The grammar might include a rule that allows for a missing semicolon and provides a way to recover from it.

**Advantages:**

* Can handle specific common errors very well.
* Integrates error handling into the normal parsing process.

**Disadvantages:**

* Makes the grammar more complex.
* Only works for errors anticipated by the grammar writer.

**4. Global Correction:**

**How it works:** The parser tries to find the smallest set of changes that will make the entire input correct.

**Example:** If there are multiple missing semicolons, the parser determines the best places to add them to fix the code.

**Advantages:**

* Produces very accurate and minimal corrections.
* Good for understanding and correcting multiple errors.

**Disadvantages:**

* Very complex and computationally expensive.
* Can be slow, especially for large inputs.

**Comparison and Opinion**

**Which one is better?**

It depends on what you need:

* **For simplicity and speed,** panic mode recovery is often best. It's easy to implement and quickly moves past errors.
* **For more precise error handling,** phrase-level recovery is better. It corrects errors without skipping too much code.
* **For handling specific common errors,** error productions are useful. They handle expected errors well.
* **For the most accurate corrections**, global correction is the best, but it's complex and slow.

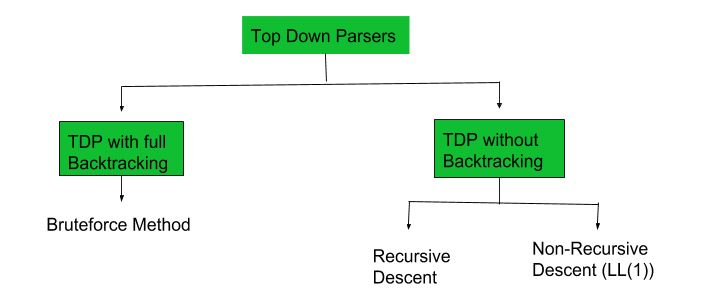
In my opinion, phrase-level recovery strikes a good balance for most practical uses. It provides precise error handling without being too complex or slow, making it suitable for many real-world applications. However, for very complex and error-prone code, global correction might be worth the extra complexity.

# **Top-down parser**

A **top-down parser** builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

* Top-Down Parser with Backtracking
* Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



**Top Down Parsers**

In this article, we are going to discuss Non-Recursive Descent which is also known as LL(1) Parser.

**LL(1) Parsing:** Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

**1. Recursive Predictive Descent Parser :**

Recursive Descent Parser is a top-down method of syntax analysis in which a set of recursive procedures is used to process input. One procedure is associated with each non-terminal of a grammar. Here we consider a simple form of recursive descent parsing called Predictive Recursive Descent Parser, in which look-ahead symbol unambiguously determines flow of control through procedure body for each non-terminal. The sequence of procedure calls during analysis of an input string implicitly defines a parse tree for input and can be used to build an explicit parse tree, if desired. In recursive descent parsing, parser may have more than one production to choose from for a single instance of input there concept of backtracking comes into play.

**Back-tracking –**

It means, if one derivation of a production fails, syntax analyzer restarts process using different rules of same production. This technique may process input string more than once to determine right production.Top- down parser start from root node (start symbol) and match input string against production rules to replace them (if matched).

To understand this, take following example of CFG :

S -> aAb | aBb

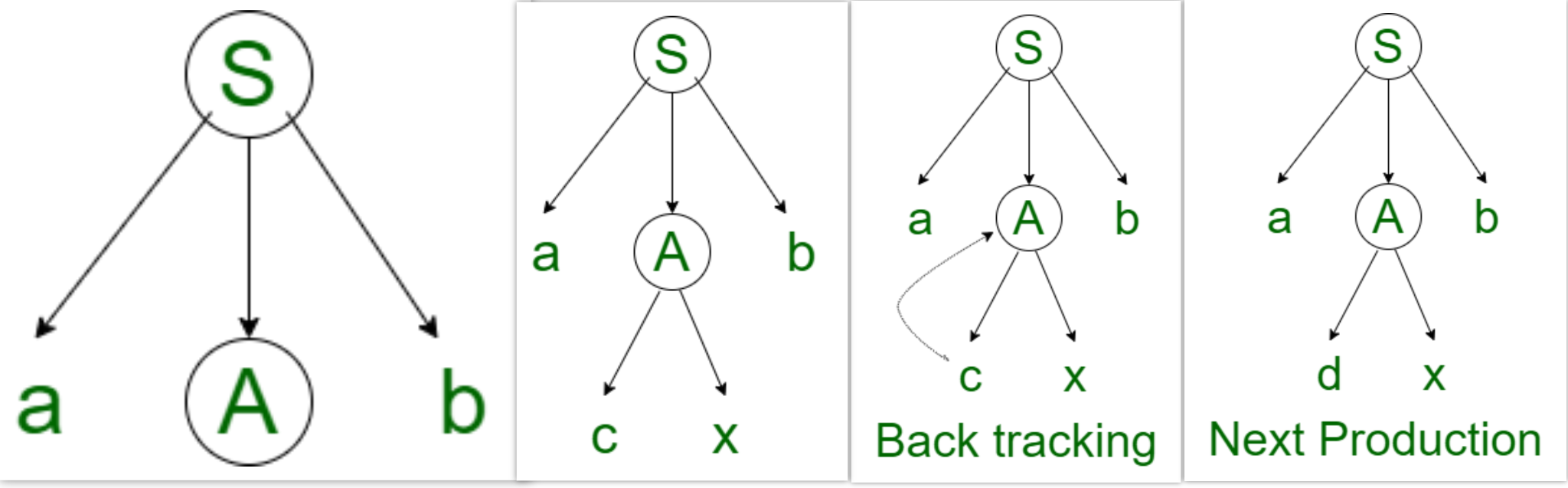
A -> cx | dx

B -> xe

For an input string – read, a top-down parser, will behave like this.

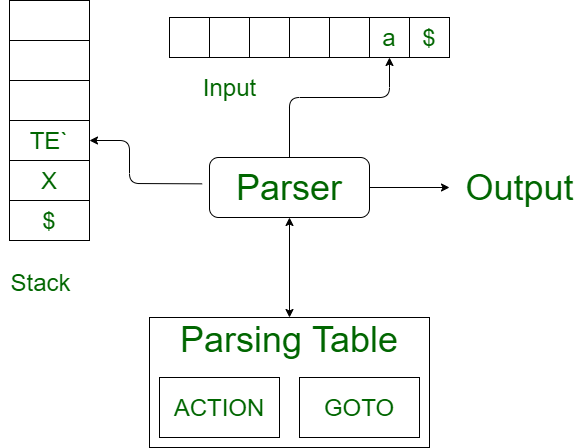
It will start with S from production rules and will match its yield to left-most letter of input, i.e. ‘a’. The very production of S (S -> aAb) matches with it. So top-down parser advances to next input letter (i.e., ‘d’). The parser tries to expand non-terminal ‘A’ and checks its production from left (A -> cx). It does not match with next input symbol. So top-down parser backtracks to obtain next production rule of A, (A -> dx).

Now parser matches all input letters in an ordered manner. The string is accepted.



**2. Non-Recursive Predictive Descent Parser :**

A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. It is also called as LL(1) parsing table technique since we would be building a table for string to be parsed. It has capability to predict which production is to be used to replace input string. To accomplish its tasks, predictive parser uses a look-ahead pointer, which points to next input symbols. To make parser back-tracking free, predictive parser puts some constraints on grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse input and generate a parse tree. Both stack and input contains an end symbol $ to denote that stack is empty and input is consumed. The parser refers to parsing table to take any decision on input and stack element combination. There might be instances where there is no production matching input string, making parsing procedure to fail.