|  |  |  |
| --- | --- | --- |
| **Factor** | **Abstract Syntax Tree** | **Parse Tree** |
| Structure | An AST is composed of symbols, such as operators, identifiers, and keywords. These symbols are organized in a tree structure. | A parse tree is composed of tokens, such as terminal and non-terminal symbols. These tokens are organized in a tree structure. |
| Representation | An AST is represented using a standard tree structure, with each node representing a particular symbol or rule. | A parse tree is represented using a graph structure, with each node representing a particular token or rule. |
| Generation | An AST is generated by the compiler from the source code. | A parse tree is generated by the parser from the source code. |
| Usage | An AST is used to analyze the source code and optimize it for compilation. | A parse tree is used to check the syntactic correctness of the source code. |
| Expressiveness | An AST is more expressive than a parse tree, as it provides more information about the source code. | A parse tree is less expressive than an AST, as it only provides basic information about the source code. |
| Efficiency | An AST is more efficient than a parse tree, as it requires fewer operations to generate. | A parse tree is less efficient than an AST, as it requires more operations to generate. |
| Size | An AST is smaller than a parse tree, as it contains fewer symbols. | A parse tree is larger than an AST, as it contains more tokens. |
| Complexity | An AST is simpler than a parse tree, as it is generated from fewer operations. | A parse tree is more complex than an AST, as it is generated from more operations. |
| Comprehension | An AST is easier to comprehend than a parse tree, as it contains more information about the source code. | A parse tree is harder to comprehend than an AST, as it contains less information about the source code. |
| Redundancy | Does not include redundant data. | Can contain duplicate or redundant information. |
| Relationship | Cannot be directly converted back into a complete parse tree due to lost information. | Can be transformed into an AST by eliminating redundancy and restructuring. |
| Purpose | Used for various tasks like code analysis, optimization, or generating machine code. | Serves as an intermediate representation during parsing, used by the compiler to verify code adheres to grammar rules. |

|  |  |  |
| --- | --- | --- |
| Sr.No. | Synthesized Attributes | Inherited Attributes |
| 1. | An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes. | An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node. |
| 2. | The production must have non-terminal as its head. | The production must have non-terminal as a symbol in its body. |
| 3. | A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself. | A Inherited attribute at node n is defined only in terms of attribute values of n’s parent, n itself, and n’s siblings. |
| 4. | It can be evaluated during a single bottom-up traversal of parse tree. | It can be evaluated during a single top-down and sideways traversal of parse tree. |
| 5. | Synthesized attributes can be contained by both the terminals or non-terminals. | Inherited attributes can’t be contained by both, It is only contained by non-terminals. |
| 6. | Synthesized attribute is used by both S-attributed SDT and L-attributed SDT. | Inherited attribute is used by only L-attributed SDT. |
| 7. | A synthesized attribute for a non-terminal node is typically unique. There's usually one value being computed and passed upwards. | Inherited attributes for a non-terminal node can be multiple. A node might inherit different context information from its parent and siblings. |
| 8. | Synthesized attribute value depends on the attribute values of its child nodes in the parse tree. | Inherited attribute value depends on the attribute values of its parent and potentially its siblings. |
| 9. | To illustrate, assume the following production S → ABC if S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S. | In case of S → ABC if A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B then S is said to be Inherited Attribute. |
| 10. |  |  |

**What is Left Recursion? what is the Left Recursion for E → E + T | T T → T \* F | F F → (E) | id**

Left recursion occurs when a grammar rule defines a non-terminal (a variable representing a language construct) in a way that creates an infinite loop for the parser. Imagine a non-terminal called expr that represents an expression in a programming language.

Here's an example of left recursion in a grammar rule for expr:

expr -> expr + term // Left-recursive rule

This rule says an expression can be formed by another expression followed by a plus sign and a term. Seems straightforward, right? Not quite.

The issue arises because the parser works by recursively applying grammar rules. In this case, when the parser encounters an expr, it tries to understand it using the available rules. Here's the problem:

The parser sees the rule expr -> expr + term.

It attempts to understand the first expr by applying the same rule again.

This leads back to step 1, creating an infinite loop.

The parser keeps calling itself on the same rule (expr -> expr + term) without ever making progress on the actual expression because the first element it encounters is itself. It's like trying to climb an infinitely repeating staircase.

Key Points:

Left recursion specifically affects recursive-descent parsers, a popular parsing technique used in compilers.

It creates an infinite loop due to the parser's recursive nature.

It hinders the parser's ability to make progress and correctly understand the code structure.

Consequences of Left Recursion

Left recursion can lead to several problems in compiler design:

Infinite Looping: As explained earlier, the parser gets stuck in a loop, wasting resources and potentially crashing the compiler.

Inefficiency: Even if the loop is detected, parsing left-recursive grammars can be inefficient compared to non-recursive ones.

Ambiguity: In some cases, left recursion can introduce ambiguity in the parsing process, making it unclear how the code should be interpreted.

Eliminating Left Recursion

The good news is that left recursion can be eliminated from context-free grammars. Here's the general approach:

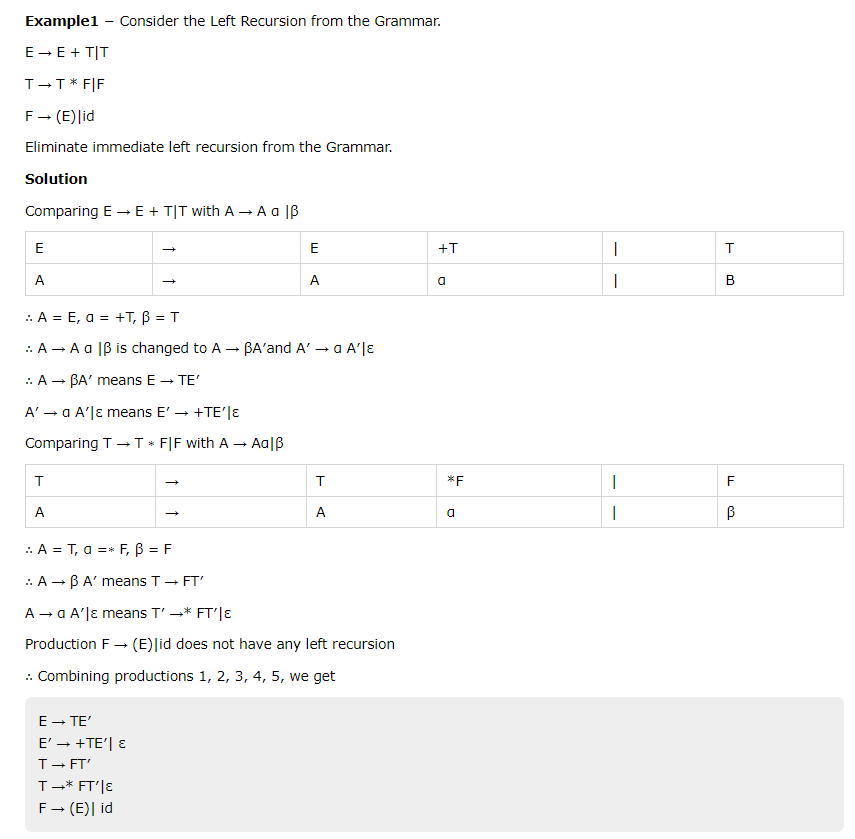
Identify left-recursive rules: Look for non-terminals appearing as the first element on the right-hand side of their own production rule.

Introduce a new non-terminal: Create a new non-terminal symbol to represent the "non-recursive" part of the left-recursive rule.

Rewrite the rules: Split the left-recursive rule into two separate rules:

One rule defines the new non-terminal using the "non-recursive" part of the original rule.

The original non-terminal's rule now references the new non-terminal, followed by an optional repetition



**Explain following Error-Recovery Strategies:**

**1.Panic Mode**

**2.Phrase-Level Recovery**

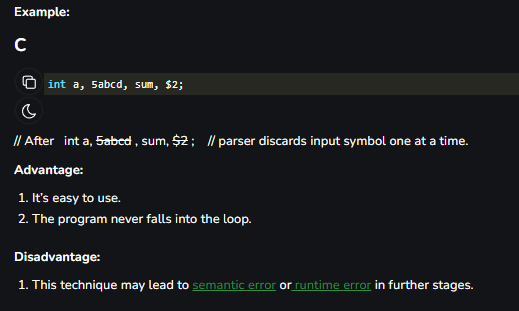
**3.Error Productions**

**4.Global Correction**

**Panic-Mode Recovery**

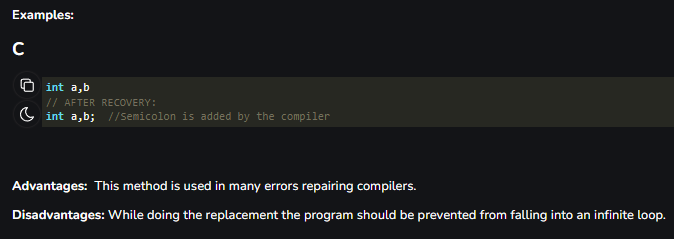
With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer

must select the synchronizing tokens appropriate for the source language. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and, unlike some methods to be considered later, is guaranteed not to go into an innite loop.



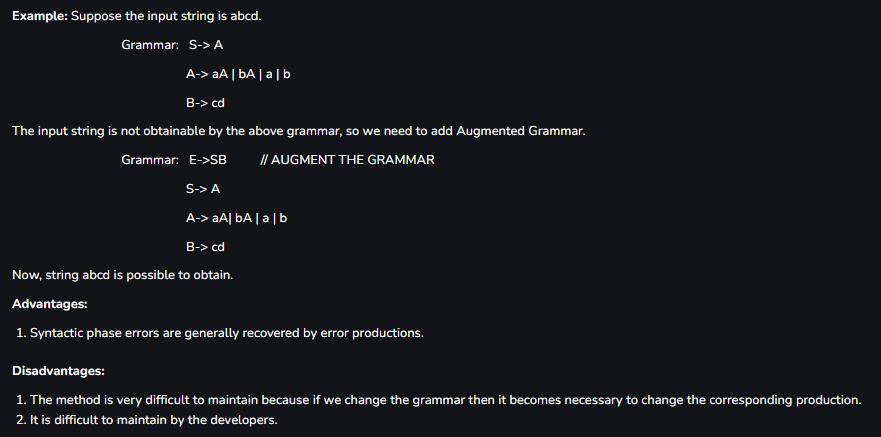
**Phrase-Level Recovery :**

On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a pre x of the remaining input by some string that allows the parser to continue. Atypical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer. Of course, wemust be careful to choose replacements that do not lead to innite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol. Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the di culty it has in coping with situations in which the actual error has occurred before the point of detection.



**Error Productions :**

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input. Global Correction Ideally,wewould like a compiler to make as few changesas possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will nd a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. 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. Donote that a closest correct programmay not be what the programmerhad in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for nding optimal replacement strings for phrase-level recovery.



**Global Correction**

Ideally,wewould like a compiler to make as few changesas possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will nd a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. 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. Donote that a closest correct programmay not be what the programmerhad in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for nding optimal replacement strings for phrase-level recovery.

