**What are Tokens?**

Tokens are fundamental units of a programming language, similar to words in a sentence. They are the basic building blocks that the compiler understands and processes. Examples of tokens include:

* Keywords (e.g., int, if, while)
* Identifiers (user-defined names like variables)
* Operators (+, -, \*, /)
* Delimiters (punctuation like comma, semicolon)
* Literals (constants like numbers and strings)

**Recognition of Tokens:**

The process of recognizing tokens from a stream of characters is called lexical analysis. This is typically done by a component called a lexical analyzer or scanner. Here are some common techniques used for token recognition:

1. **Transition Diagrams:**
   * These are graphical representations of states and transitions that a scanner undergoes while reading characters.
   * Each state represents a potential token or a point in the analysis process.
   * Transitions between states occur based on the next character encountered.
   * Reaching a specific final state confirms a token's recognition.
2. **Recognition of Reserved Words and Identifiers:**
   * Reserved words are pre-defined keywords with specific meanings in the language (e.g., "if", "for").
   * They are often stored in a dictionary and checked against the input stream for exact matches.
   * Identifiers are user-defined names that follow specific naming rules (e.g., starting with a letter).
   * Regular expressions are often used to validate their structure.
3. **Completion of the Running Example:**
   * This refers to the process of finalizing the token recognition once a complete token is identified.
   * This may involve associating a specific attribute with the token (e.g., its type and value).
   * The scanner then moves on to the next character in the stream.
4. **Architecture of a Transition-Diagram-Based Lexical Analyzer:**
   * The scanner typically consists of a finite state machine represented by the transition diagram.
   * It has an input buffer to hold the characters being read.
   * It maintains a current state and transitions based on the next character.
   * Upon reaching a final state, the token is recognized, and the process repeats for the next character.

**Specification of Tokens:**

Specifying tokens involves defining the rules that govern their structure and recognition. Here are some key concepts involved:

1. **Strings and Languages:**
   * A string is a sequence of characters.
   * A language is a set of strings that follow specific rules.
2. **Operations on Languages:**
   * Operations like union, intersection, and concatenation are used to combine and manipulate sets of strings.
   * These operations help define the rules for valid token structures.
3. **Regular Expressions:**
   * Regular expressions are powerful tools for specifying patterns of characters that define valid tokens.
   * They use special symbols and operators to represent character sequences and their repetition.
4. **Regular Definitions:**
   * Regular definitions are formal rules that combine regular expressions and operations to define the structure of different token types.
   * These definitions are used by the lexical analyzer to identify and categorize tokens during the analysis process.

Explain structure of Lexical Analyzer.

1. What is syntax analysis? Explain structure of a parser.

* When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.
* Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis.
* It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
* It does so by building a data structure, called a Parse tree or Syntax tree.
* The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
* If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.
* Syntax analysis, also known as parsing, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language. This is typically the second stage of the compilation process, following lexical analysis.
* The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.

**The syntax analysis phase typically involves the following steps:**

* **Tokenization:** The input program is divided into a sequence of tokens, which are basic building blocks of the programming language, such as identifiers, keywords, operators, and literals.
* **Parsing:** The tokens are analyzed according to the grammar rules of the programming language, and a parse tree or AST is constructed that represents the hierarchical structure of the program.
* **Error handling:** If the input program contains syntax errors, the syntax analyzer detects and reports them to the user, along with an indication of where the error occurred.
* **Symbol table creation:** The syntax analyzer creates a symbol table, which is a data structure that stores information about the identifiers used in the program, such as their type, scope, and location.

**Advantages:**

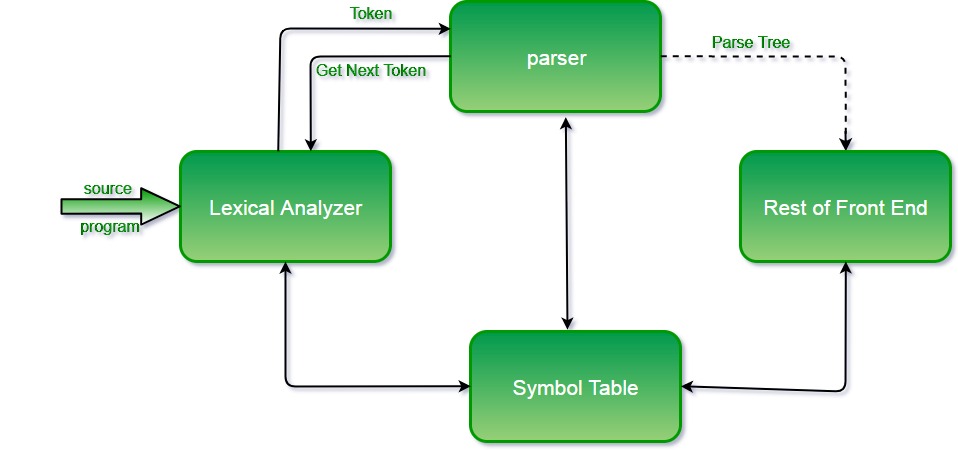
Advantages of using syntax analysis in compiler design include:

1. **Structural validation:** Syntax analysis allows the compiler to check if the source code follows the grammatical rules of the programming language, which helps to detect and report errors in the source code.
2. **Improved code generation:** Syntax analysis can generate a parse tree or abstract syntax tree (AST) of the source code, which can be used in the code generation phase of the compiler design to generate more efficient and optimized code.
3. **Easier semantic analysis:** Once the parse tree or AST is constructed, the compiler can perform semantic analysis more easily, as it can rely on the structural information provided by the parse tree or AST.

**Disadvantages:**

Disadvantages of using syntax analysis in compiler design include:

1. **Complexity:** Parsing is a complex process, and the quality of the parser can greatly impact the performance of the resulting code. Implementing a parser for a complex programming language can be a challenging task, especially for languages with ambiguous grammars.
2. **Reduced performance:** Syntax analysis can add overhead to the compilation process, which can reduce the performance of the compiler.
3. **Limited error recovery:** Syntax analysis algorithms may not be able to recover from errors in the source code, which can lead to incomplete or incorrect parse trees and make it difficult for the compiler to continue the compilation process.
4. **Inability to handle all languages:** Not all languages have formal grammars, and some languages may not be easily parseable.



Parsing is performed at the syntax analysis phase where a stream of tokens is taken as input from the lexical analyzer and the parser produces the parser tree for the tokens while checking the stream of tokens against the syntax errors.

**Role of Parser**

* In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser.
* The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language.
* It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

**Types of Parsing**

The parsing is divided into two types, which are as follows:

**1. Top-down Parsing**

Top-down parsing attempts to build the parse tree from the root node to the leaf node. The top-down parser will start from the start symbol and proceed to the string. It follows the leftmost derivation. In leftmost derivation, the leftmost non-terminal in each sentential is always chosen.

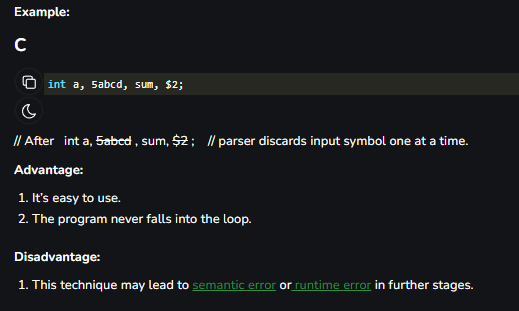
**2. Bottom-up Parsing :**

Bottom-up parsing builds the parse tree from the leaf node to the root node. The bottom-up parsing will reduce the input string to the start symbol. It traces the rightmost derivation of the string in reverse. Bottom-up parsers are also known as shift-reduce parsers.

1. **Explain syntax errors and their recovery with the help of compilers.**
2. **Explain following error recovery strategies:**
3. **Panic-Mode Recovery**

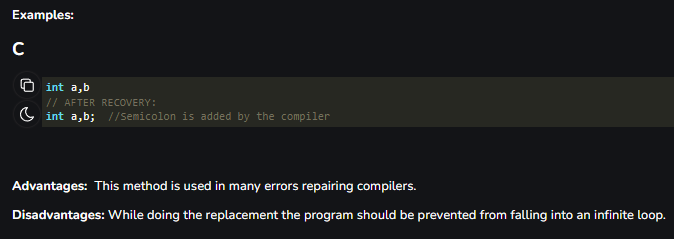
 With this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous. The compiler designer

 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.



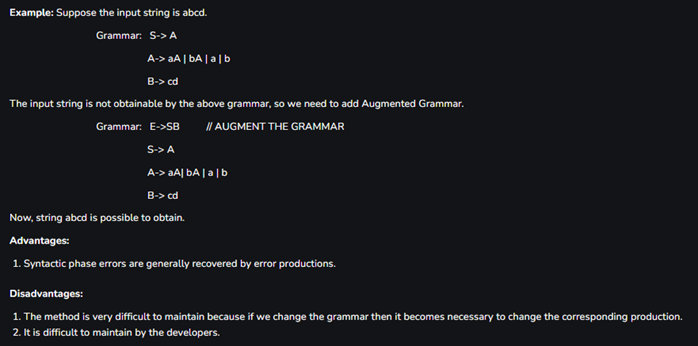
**2. Phrase-Level Recovery :**

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.



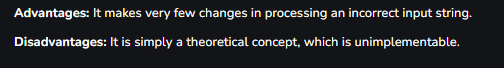
**3. Error Productions :**

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.



**4. 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.



1. What is ambiguity in grammar? Explain with example.
2. What is precedence and associativity of operators? Explain with example.
3. What is top down? Explain with example.
4. What is recursive descent parsers? Explain with example.
5. What is backtracking? Explain with example.