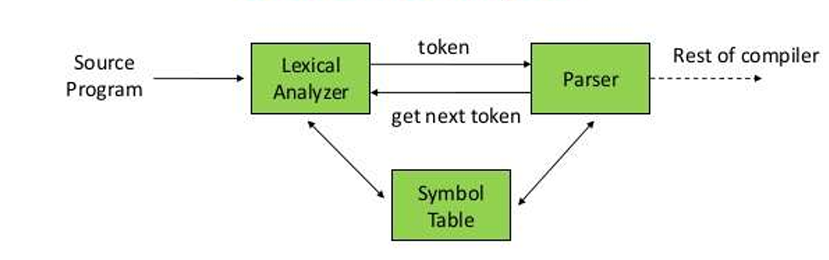
**Lexical Analysis**

**Explain structure of Lexical Analyzer.**

**Structure of Lexical Analyzer**

* Lexical Analysis is the initial phase of a compiler, also referred to as a scanner. Its purpose is to convert high-level input programs into a sequence of tokens.
* Lexical Analysis can be implemented using Deterministic Finite Automata (DFA).
* The output of the Lexical Analyzer is a sequence of tokens sent to the parser for syntax analysis.



**What is a Token?**

* A lexical token is a sequence of characters treated as a unit in the grammar of programming languages.
* Examples include type tokens (id, number, real), punctuation tokens (IF, void, return), and alphabetic tokens (keywords).

**Lexeme:**

* A sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters comprising a single token.
* Examples include “float”, “abs\_zero\_Kelvin”, “=”, “-”, “273”, “;”.

**How Lexical Analyzer Works?**

1. **Input Preprocessing:** Clean up the input text by removing comments, whitespace, and non-essential characters.
2. **Tokenization:** Break the input text into a sequence of tokens by matching characters against predefined patterns or regular expressions.
3. **Token Classification:** Determine the type of each token such as keywords, identifiers, operators, and punctuation symbols.
4. **Token Validation:** Check each token for validity according to the rules of the programming language.
5. **Output Generation:** Generate the output of the lexical analysis process, typically a list of tokens for further processing.

**Error Detection:**

* The lexical analyzer identifies errors using automation machine and grammar of the given language, providing row and column numbers of the error.

**Token Sequence Example:**

* For a statement like **a = b + c;**, the token sequence is **id=id+id;**, with each **id** referencing its variable in the symbol table.

**Advantages:**

* **Simplifies Parsing:** Breaking down the source code into tokens makes it easier for computers to understand and work with the code.
* **Error Detection:** Detects lexical errors early in the compilation process, improving overall efficiency.
* **Efficiency:** Once source code is tokenized, subsequent phases of compilation or interpretation operate more efficiently.

**Disadvantages:**

* **Limited Context:** Lexical analysis operates on individual tokens, potentially leading to ambiguity in complex languages.
* **Overhead:** Adds computational overhead to the compilation process.
* **Debugging Challenges:** Lexical errors may not always provide clear indications of their origins, posing challenges during debugging.

**How error recovery is handled in lexical analysis?**

**Error Recovery in Lexical Analysis**

1. **Error Recovery Technique:**
   * When the lexical analyzer encounters a situation where none of the patterns for tokens matches any prefix of the remaining input, error recovery becomes necessary.
   * The simplest recovery strategy is called "panic mode" recovery. It involves deleting successive characters from the remaining input until the lexical analyzer can identify a well-formed token at the beginning of what remains.
2. **Error-Recovery Actions:**
   * Transpose of two adjacent characters.
   * Insert a missing character into the remaining input.
   * Replace a character with another character.
   * Delete one character from the remaining input.
3. **Handling Errors in Lexical Analysis:**
   * **Leaving the Statement:** The lexical analyzer may identify the error, correct it internally, and then proceed to call the parser for syntax analysis.
   * **No Matching Pattern:** In the event of no matching pattern, the simplest recovery strategy, "panic mode recovery," is employed. This involves not processing the remaining characters. However, this approach is not ideal as tokens still need to be sent to the parser for syntax analysis.
4. **Other Possible Recovery Methods:**
   * Inserting a missing character.
   * Replacing an incorrect character with the correct one.
   * Transposing (changing the place of) adjacent characters.
   * Deleting an extra character.
5. **Goal of Error Recovery:**
   * The primary goal of error recovery in lexical analysis is to recover from errors and continue the lexical analysis process so that tokens can be passed to the parser for further processing.
   * Simply abandoning the input is not a viable solution as it would hinder the parser from receiving the necessary tokens.

**What are tokens? Explain specification & recognition of tokens.**

**Tokens: Specification & Recognition**

* In programming languages, tokens represent various lexical units such as keywords, constants, identifiers, strings, numbers, operators, and punctuation symbols.
* For example, in C language, the line **int value = 100;** contains tokens like **int** (keyword), **value** (identifier), **=** (operator), **100** (constant), and **;** (symbol).

**Specifications of Tokens:**

1. **Alphabets & Strings:**
   * Alphabets are finite sets of symbols used in languages.
   * Strings are finite sequences of alphabets.
2. **Special Symbols:**

Programming languages consist of various symbols like arithmetic symbols, punctuation, assignment symbols, comparison symbols, and preprocessor directives.

1. **Language:**

A language is a finite set of strings over a finite set of alphabets.

1. **Longest Match Rule:**

The lexical analyzer follows the longest match rule, wherein it identifies tokens until it encounters whitespace, operator symbols, or special symbols.

1. **Operations & Notations:**

Operations on languages include union, concatenation, and Kleene closure, denoted by regular expressions.

1. **Finite Automata:**

Finite automata are state machines that process input symbols and transition between states. They consist of states, input symbols, start state, final states, and a transition function.

**Some key operations on regular expressions include:**

* Union: (r)|(s) is a regular expression denoting the language L(r) ∪ L(s).
* Concatenation: (r)(s) is a regular expression denoting the language L(r)L(s).
* Kleene closure: (r)\* is a regular expression denoting the language (L(r))\*.

**The precedence and associativity of these operations are as follows:**

* \* has the highest precedence.
* Concatenation . has the second highest precedence.
* | (pipe) has the lowest precedence.
* All operators are left-associative.

**Recognition of Tokens:**

1. **Finite Automata:**
   * Recognition of tokens is typically performed using finite automata, which represent the actions of the lexical analyzer.
   * The lexical analyzer reads input from the input buffer, starting from the ‘***lexemeBegin***’ pointer and advancing using a forward pointer.
   * It calculates the set of states at each point based on the input symbols.
   * If no next state is available for an input symbol, the analyzer determines the longest prefix that matches a pattern.
   * This process continues until one or more accepting states are reached.
2. **Decision Making:**

If multiple accepting states are reached, the pattern that appears earliest in the list of the Lex program is chosen.

**Explain following error recovery strategies:**

* 1. **Panic Mode**
  2. **Phrase Level**
  3. **Error Productions**
  4. **Global Corrections**

1. **Panic Mode:**

* The panic mode is the simplest error recovery strategy. When an error is encountered, the parser discards the remaining input until it reaches a synchronization point, such as the end of the current statement.
* This approach does not attempt to correct the error but rather skips over it until a point where parsing can resume normally.

**Advantage:**

* It’s easy to use.
* The program never falls into the loop.

**Disadvantage:**

* This technique may lead to [semantic error](https://www.geeksforgeeks.org/errors-in-cc/) or[runtime error](https://www.geeksforgeeks.org/runtime-errors/) in further stages.

1. **Phrase Level:**

* Phrase level error recovery aims to resynchronize the parser by finding the end of the current phrase or statement.
* The parser discards input symbols until it reaches a point where it can resume parsing, typically at the end of the current statement.
* This strategy is more sophisticated than panic mode as it attempts to recover to a meaningful point in the input.

**Advantages:**

This method is used in many errors repairing compilers.

**Disadvantages:**

While doing the replacement the program should be prevented from falling into an infinite loop.

1. **Error Productions:**

* Error productions are special grammar rules added to handle syntax errors.
* When an error occurs, the parser can use these error productions to insert, delete, or replace symbols to resynchronize the parse.
* This allows the parser to continue processing the input, rather than aborting at the first error encountered.

**Advantages:**

1. Syntactic phase errors are generally recovered by error productions.

**Disadvantages:**

1. The method is very difficult to maintain because if we change the grammar then it becomes necessary to change the corresponding production.
2. It is difficult to maintain by the developers.
3. **Global Corrections:**

* Global correction strategies aim to comprehensively analyze the input and the encountered errors.
* After analysis, a series of corrections are made to the input to fix all errors at once.
* This approach requires sophisticated error analysis and correction algorithms, making it more advanced than other strategies.

**Advantages:**

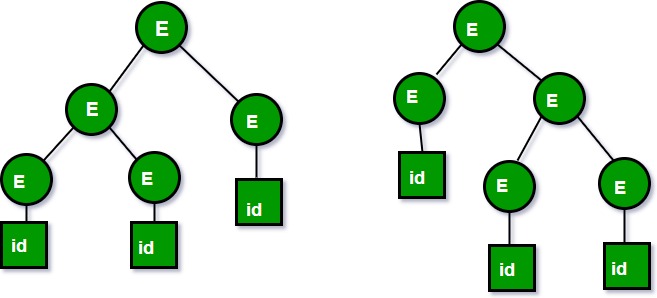
It allows basic type conversion, which we generally do in real-life calculations.

**Disadvantages:**

Only Implicit type conversion is possible.**What is ambiguity in grammar? Explain with example.**

* A Context-Free Grammar (CFG) is considered ambiguous if there exists more than one derivation tree for a given input string. This means there are multiple LeftMost Derivation Trees (LMDT) or RightMost Derivation Trees (RMDT) for the same input string.

**Example 1:** Consider the grammar:



2. Let us now consider the following grammar:

Set of alphabets ? = {0,…,9, +, \*, (, )}

E -> I

E -> E + E

E -> E \* E

E -> (E)

I -> ? | 0 | 1 | … | 9

From the above grammar String **3\*2+5** can be derived in 2 ways:

I) First leftmost derivation II) Second leftmost derivation

E=>E\*E E=>E+E

=>I\*E =>E\*E+E

=>3\*E+E =>I\*E+E

=>3\*I+E =>3\*E+E

=>3\*2+E =>3\*I+E

=>3\*2+I =>3\*2+I

=>3\*2+5 =>3\*2+5

Both the above parse trees are derived from the same grammar rules but both parse trees are different. Hence the grammar is ambiguous

**Inherently Ambiguous Language:**

* An inherently ambiguous language is one where every CFG representing the language is ambiguous. This means there is no way to rewrite the grammar to remove ambiguity.

**Questions on Ambiguous Grammar:**

1. **Determining Ambiguity:**
   * If a grammar exhibits both left and right recursion, it is likely ambiguous. For example, **S -> SaS | ?** is ambiguous because it contains both left and right recursion.
2. **Unambiguity despite no Left/Right Recursion:**
   * The absence of left or right recursion in a grammar does not guarantee unambiguity. For instance, **S -> aB | ab**, **A -> AB | a**, **B -> Abb | b** is ambiguous despite lacking left or right recursion.
3. **Identifying Ambiguity:**
   * For a grammar like **S -> SAB | ?**, we can observe that by replacing **B -> AS** in **S -> SAB**, we get **S -> SAAS**, showing both left and right recursion. Thus, the grammar is ambiguous.

Ambiguity in grammar can lead to multiple interpretations of the same input string, which is undesirable in programming languages as it can result in unexpected behavior. Identifying and resolving ambiguity is crucial for ensuring the correctness and predictability of language parsing.

**What is precedence & associativity of operators? Explain with example.**

**Precedence**

Precedence refers to the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. For example, in the expression 3 + 4 \* 5, the multiplication operator \* has higher precedence than the addition operator +, so 4 \* 5 is evaluated first, resulting in 20, and then 3 + 20 is evaluated to give 23.

**Associativity**

Associativity determines the order in which operators of the same precedence are evaluated. Operators can be left-associative, right-associative, or non-associative.

* **Left-associative**: Operators are evaluated from left to right. For example, in a - b - c, the subtraction operator is left-associative, so a - b is evaluated first, and then the result is subtracted from c.
* **Right-associative**: Operators are evaluated from right to left. For example, the exponentiation operator ^ is right-associative, so a ^ b ^ c is evaluated as a ^ (b ^ c).
* **Non-associative**: Operators do not associate in any direction. For example, the comparison operator == is non-associative, so expressions like a == b == c are not valid.

**Example**

Consider the expression 3 + 4 \* 5 / 2. The operators in this expression have different precedence levels:

1. Multiplication \* and division / have higher precedence than addition +.
2. Multiplication and division have the same precedence and are left-associative.
3. Addition has lower precedence than multiplication and division.

**Following the precedence and associativity rules:**

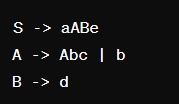
1. 4 \* 5 is evaluated first, resulting in 20.
2. 20 / 2 is evaluated next, resulting in 10.
3. Finally, 3 + 10 is evaluated, giving the final result of 13.

**What is top down parsing? Explain with example.**

* Top-down parsing is a parsing technique where the parser starts with the start symbol of the grammar and attempts to transform it into the input string by repeatedly applying production rules. It proceeds by starting at the root of the parse tree and works its way down to the leaves.
* It follows a leftmost derivation strategy, meaning it expands the leftmost nonterminal in the current sentential form during each step of parsing.

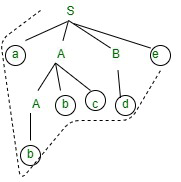
**Working of Top-Down Parsing:**

* Consider the grammar:



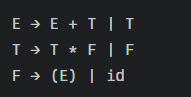
* Let's parse the input string **abbcde$** using the top-down approach.

1. Begin with the start symbol **S** and the input **abbcde$**.
2. Expand **S -> aABe**, where **a** is consumed from the input.
3. Now, the sentential form is **aABe** and the input is **bbcde$**.
4. Expand **A -> Abc**, where **b** is consumed from the input.
5. Expand **A -> bc**, where **b** is consumed from the input.
6. Now, the sentential form is **abc** and the input is **cde$**.
7. Expand **B -> d**, where **d** is consumed from the input.
8. Now, the sentential form is **abcde** and the input is **$**.
9. Expand **e**, where **$** is consumed from the input.
10. The input string has been completely consumed, and the parse tree is successfully constructed.



**Example of LL(1) Parsing:**

* Consider the grammar:



* Let's parse the input string **id + id \* id** using LL(1) parsing.

1. Start with the start symbol **E**.
2. Expand **E -> T**.
3. Expand **T -> F**.
4. Expand **F -> id**.
5. Consume **id**.
6. Consume **+**.
7. Expand **E -> E + T**.
8. Expand **T -> F**.
9. Expand **F -> id**.
10. Consume **id**.
11. Consume **\***.
12. Expand **T -> T \* F**.
13. Expand **F -> id**.
14. Consume **id**.
15. The input string has been completely consumed, and the parse tree is successfully constructed.