UNIT-IV: Machine Independent Phases (ATCD)

INTRODUCTION

Machine-independent phases, also known as front-end phases, are stages in the compilation process of a programming language that are independent of the specific target machine or architecture. These phases focus on analyzing and transforming the source code into an intermediate representation that can be optimized and subsequently translated to machine code. The machine-independent phases typically include:

1. Lexical Analysis:
- This phase involves breaking the source code into a sequence of tokens or lexemes, such as keywords, identifiers, operators, and literals.
- A lexical analyzer, also known as a scanner, performs this task by using regular expressions or finite automata to recognize the lexical patterns.

2. Syntax Analysis:
- The syntax analysis phase, also called parsing, examines the structure of the program according to a formal grammar or language syntax.
- It constructs a parse tree or an abstract syntax tree (AST) that represents the hierarchical structure of the program.

3. Semantic Analysis:
- Semantic analysis focuses on checking the meaning and correctness of the program beyond its syntax.
- It performs various checks, such as type checking, scope resolution, and identifier usage, to ensure that the program follows the language's semantics and rules.

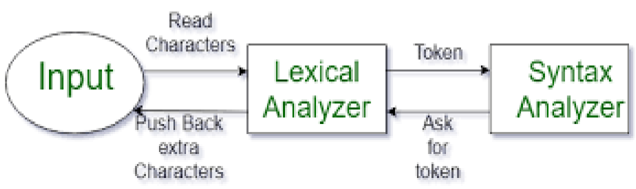
4. Intermediate Code Generation:
- In this phase, an intermediate representation of the program is generated, which is independent of the target machine.
- The intermediate code may take the form of three-address code, quadruples, or any other suitable representation that captures the essential operations of the program.

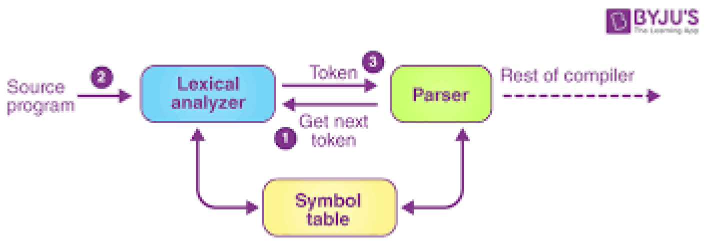
5. Optimization:
- The optimization phase aims to improve the intermediate code by applying various transformations to enhance performance, reduce code size, or improve the execution efficiency.
- Common optimization techniques include constant folding, loop optimization, dead code elimination, and register allocation.

The machine-independent phases lay the groundwork for subsequent machine-dependent phases, which focus on translating the intermediate representation into machine code specific to the target architecture. The output of the machine-independent phases serves as input for the optimization and code generation stages tailored to the target machine.

Lexical Analysis:

Lexical analysis, also known as scanning, is the first phase of the compiler that breaks the source code into a sequence of tokens or lexemes. It performs the task of identifying and categorizing the individual meaningful units in the source code, such as keywords, identifiers, operators, literals, and punctuation symbols. The resulting tokens are then used as input for the subsequent phases of the compilation process.





The process of lexical analysis involves the following steps:

1. Tokenization:
- The source code is read character by character.
- The lexical analyzer identifies groups of characters that form meaningful units called tokens.
- Tokens represent the smallest meaningful units in the language, such as keywords, identifiers, and operators.

2. Lexical Rules:
- The lexical analyzer applies a set of predefined lexical rules to determine the category of each token.
- Lexical rules are typically defined using regular expressions or finite automata.
- Regular expressions describe the patterns that tokens should match, and the lexical analyzer applies these patterns to recognize tokens.

3. Ignoring Whitespace and Comments:
- The lexical analyzer skips over whitespace characters (e.g., spaces, tabs, and newlines) and removes them from consideration as tokens.
- Comments, both single-line and multi-line, are also ignored by the lexical analyzer.

4. Building Symbol Tables:
- The lexical analyzer may construct a symbol table, which is a data structure that keeps track of identifiers and their associated attributes.
- As identifiers are encountered in the source code, they are added to the symbol table for later use in the semantic analysis phase.

5. Error Handling:
- The lexical analyzer reports any lexical errors encountered, such as unrecognized symbols or invalid tokens.
- It may provide error messages indicating the line number and the nature of the error to aid in debugging.

The output of the lexical analysis phase is a stream of tokens, each representing a different category of lexeme in the source code. These tokens serve as input for the subsequent phases of the compiler, such as syntax analysis and semantic analysis.

Logical phases of compiler

The logical phases of a compiler, also known as the high-level phases, are the major steps involved in the compilation process. These phases are organized in a logical order and collectively perform the transformation of the source code into executable machine code or an equivalent form.

The main logical phases of a compiler are as follows:

1. Lexical Analysis:
- The source code is divided into a sequence of tokens or lexemes, representing the basic units of the language.
- Lexical analysis involves tasks such as tokenization, recognizing keywords and identifiers, and discarding whitespace and comments.

2. Syntax Analysis:
- The sequence of tokens is analyzed based on the language's grammar rules to determine the program's syntactic structure.
- The syntax analysis phase builds a parse tree or an abstract syntax tree (AST) that represents the hierarchical structure of the program.

3. Semantic Analysis:
- Semantic analysis focuses on checking the meaning and correctness of the program beyond its syntax.
- It performs tasks such as type checking, scope resolution, and checking for semantic errors.
- The semantic analysis phase ensures that the program follows the language's semantics and rules.

4. Intermediate Code Generation:
- The compiler generates an intermediate representation (IR) of the program.
- The IR is a platform-independent representation that captures the essential operations and structure of the program.
- Intermediate code generation prepares the program for subsequent optimization and code generation phases.

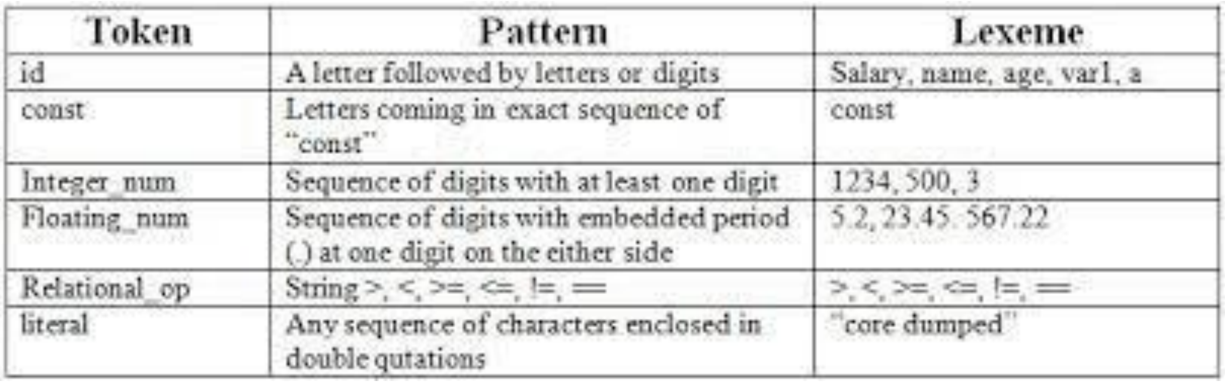
5. Optimization:
- The compiler applies various transformations to the intermediate code to improve its efficiency or reduce its size.
- Optimization techniques aim to optimize the program's performance, such as loop optimization, constant propagation, and dead code elimination.
- Optimization may involve data flow analysis, control flow analysis, and other program analysis techniques.

6. Code Generation:
- The compiler generates the target machine code or an equivalent executable form based on the intermediate representation.
- Code generation involves mapping the intermediate code to the specific instructions and memory operations of the target architecture.
- The generated code may undergo further optimizations specific to the target machine.

7. Symbol Table Management:
- Throughout the compilation process, a symbol table is maintained to store information about identifiers, their types, and scope.
- Symbol table management involves creating and updating the symbol table as identifiers are encountered and processed.

Lexemes Tokens and patterns

Lexemes, tokens, and patterns are key concepts in lexical analysis, which is the process of breaking down the source code into meaningful units for further processing by the compiler. Let's define each of these terms:



Token	Pattern	Lexeme
id	A letter followed by letters or digits	Salary, name, age, var1, a
const	Letters coming in exact sequence of "const"	const
Integer_num	Sequence of digits with at least one digit	1234, 500, 3
Floating_num	Sequence of digits with embedded period (.) at one digit on the either side	5.2, 23.45 567.22
Relational_op	String >, <, >=, <=, !=, ==	>, <, >=, <=, !=, ==
literal	Any sequence of characters enclosed in double qutations	"core dumped"

1. Lexeme:
- A lexeme is the smallest unit of meaningful information in the source code.
- It represents a sequence of characters that forms a single entity in the programming language.
- Examples of lexemes include keywords (e.g., "if", "while"), identifiers (e.g., variable names), literals (e.g., numbers, strings), and operators (e.g., "+", "-", "=").

2. Token:

- A token is a categorized representation of a lexeme.
- Tokens are the output of the lexical analysis phase and serve as the input for the subsequent phases of the compiler.
- Each token has a name (also known as a token type) and may have associated attributes.
- Examples of tokens include keywords (e.g., "if" represented by the token type "IF"), identifiers (e.g., "x" represented by the token type "IDENTIFIER"), literals (e.g., "42" represented by the token type "INTEGER_LITERAL"), and operators (e.g., "+" represented by the token type "PLUS").

3. Pattern:

- A pattern is a description or rule that defines the lexical structure of a lexeme.
- Patterns are typically defined using regular expressions or other pattern matching mechanisms.
- Patterns specify the valid sequence of characters that a lexeme should match to be recognized as a specific token type.
- For example, a pattern for an identifier in a programming language might specify that it should start with a letter followed by a sequence of letters or digits.

In the lexical analysis phase, the source code is scanned character by character, and lexemes are identified based on the defined patterns. Each identified lexeme is then categorized as a token by assigning a token type to it. The tokens form a sequence that represents the structure of the source code and is subsequently used for further analysis and processing.

Lexical Errors

Lexical errors, also known as lexical or token-level errors, occur during the process of lexical analysis when the source code contains invalid or unrecognized tokens or lexemes. These errors indicate violations of the language's lexical rules or syntax and can prevent the code from being correctly interpreted or compiled.

Here are a few common types of lexical errors:

1. Invalid or Unrecognized Tokens:

- These errors occur when a token is encountered that does not match any valid token type defined in the language.
- Examples include misspelled keywords, using symbols or characters that are not defined in the language, or using reserved words as identifiers.

2. Missing or Unbalanced Delimiters:

- Delimiters, such as parentheses, brackets, or quotes, are used to enclose specific sections of code.
- Lexical errors can occur if these delimiters are missing or not properly balanced, meaning that the opening and closing delimiters do not match.

3. Incorrect Numeric or String Representations:

- Lexical errors can occur when numbers or strings are not represented correctly according to the language's syntax rules.
- For example, a number with an invalid format or a string missing closing quotes can lead to lexical errors.

4. Illegal Characters or Symbols:

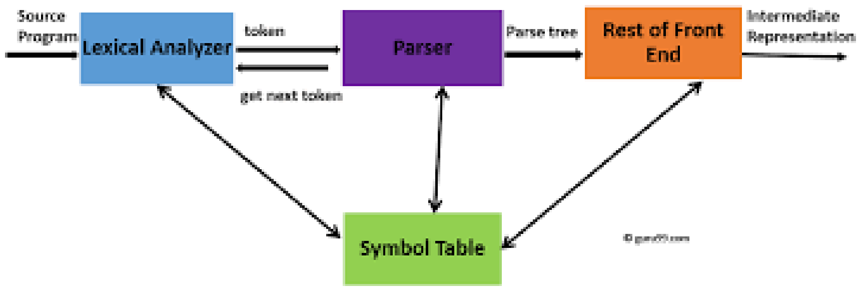
- Lexical errors can arise when the source code contains characters or symbols that are not valid in the language.
- This could include using special characters that are not allowed or using operators or symbols incorrectly.

When a lexical error is encountered, the compiler or interpreter typically generates an error message indicating the specific location of the error, such as the line number or character position. These error messages can help the developer identify and fix the issues in the source code.

Handling lexical errors is an important part of the compilation or interpretation process. The lexical analyzer is responsible for identifying and reporting these errors so that the developer can correct them before proceeding to subsequent phases of the compilation process.

Syntax Analysis:

Syntax analysis, also known as parsing, is a phase in the compilation process that follows lexical analysis. Its primary objective is to analyze the syntactic structure of the source code based on the grammar rules of the programming language. Syntax analysis ensures that the arrangement of tokens in the source code adheres to the specified grammar and identifies any syntax errors or inconsistencies.



Here are some key aspects of syntax analysis:

1. Grammar Rules:

- Syntax analysis utilizes a formal grammar that defines the syntax of the programming language.
- The grammar consists of a set of production rules that specify the valid combinations of tokens and their ordering.
- Context-free grammars, often expressed using Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF), are commonly used to describe programming language syntax.

2. Parsing Techniques:

- Syntax analysis uses parsing techniques to analyze the source code and construct a parse tree or an abstract syntax tree (AST) representing the syntactic structure.
- Common parsing techniques include top-down parsing (e.g., recursive descent) and bottom-up parsing (e.g., LR parsing).

3. Parse Tree and Abstract Syntax Tree (AST):

- A parse tree is a hierarchical representation of the syntactic structure of the source code, generated during parsing.
- It shows the derivation of the program based on the production rules of the grammar.
- An AST is a simplified version of the parse tree that retains only the essential information, removing redundant details.
- The AST focuses on the logical structure of the program and is often used for subsequent analysis and code generation.

4. Syntax Errors:

- During syntax analysis, if the source code violates the grammar rules, syntax errors are detected.
- Syntax errors indicate inconsistencies in the structure of the code, such as missing or misplaced tokens, incorrect use of operators, or invalid expressions.
- Syntax error messages typically provide information about the location and nature of the error to help developers identify and rectify the issue.

The syntax analysis phase ensures that the source code is well-formed and follows the specified grammar rules. It plays a critical role in the compilation process, as it establishes the foundation for subsequent phases such as semantic analysis, optimization, and code generation. By identifying syntax errors early on, developers can correct them and ensure that the program is correctly structured before proceeding with further compilation steps.

Parsing definition

Parsing, also known as syntactic analysis, is a process in computer science and linguistics that involves analyzing the grammatical structure of a given sequence of tokens or symbols, such as those produced by a lexical analyzer. It aims to determine whether the sequence of tokens conforms to the rules defined by a formal grammar or language syntax.

In the context of programming languages, parsing is a crucial step in the compilation process. It verifies the syntactic correctness of the source code by analyzing the arrangement and relationships of tokens based on the grammar rules. The primary goal of parsing is to construct a parse tree or an abstract syntax tree (AST), which represents the hierarchical structure and relationships between the tokens, following the rules of the grammar.**The** parsing process involves applying parsing techniques and algorithms to the token sequence and the corresponding grammar rules. These techniques determine how the tokens should be grouped and organized, leading to the construction of the parse tree or AST. <u>Parsing techniques can be categorized into two main approaches:</u>

1. Top-Down Parsing:

- Top-down parsing starts from the top of the parse tree or AST and attempts to find a sequence of production rule applications that matches the given token sequence.
- It usually employs a recursive descent approach, where each non-terminal in the grammar is associated with a parsing function or subroutine.
- The top-down parser tries to expand non-terminals in a depth-first manner, matching the tokens with the appropriate production rules until a complete parse tree or AST is constructed.

2. Bottom-Up Parsing:

- Bottom-up parsing starts from the tokens and works upwards, attempting to construct a parse tree or AST by reducing a sequence of tokens to higher-level non-terminals.
- Bottom-up parsing algorithms, such as LR (left-to-right, rightmost derivation) parsing, use tables and state machines to efficiently perform reductions and identify valid sequences of tokens that match the grammar.
- Bottom-up parsers handle a broader class of grammars than top-down parsers but may require more complex algorithms and additional processing.

Parsing can detect syntax errors when the given token sequence cannot be successfully parsed according to the grammar rules. These errors indicate violations of the language's syntax and are usually accompanied by error messages that provide information about the location and nature of the error.

types of parsers

There are several types of parsers, each employing different techniques and algorithms to analyze the structure of a given sequence of tokens or symbols. The choice of parser depends on factors such as the complexity of the grammar, efficiency requirements, and error recovery capabilities. Here are some common types of parsers:

1. Recursive Descent Parser:
- A top-down parsing technique that uses recursive procedures or functions to match non-terminals in the grammar.
- Each non-terminal corresponds to a parsing function, which recursively calls other parsing functions to handle sub-expressions.
- Recursive descent parsers are easy to understand and implement, especially for LL(k) grammars, where LL stands for left-to-right, leftmost derivation, and k denotes the number of lookahead tokens.

2. LL Parser (Table-Driven):
- An LL parser predicts and analyzes the input based on a look-ahead token.
- It uses a parsing table that stores the parsing decisions for different combinations of non-terminals and look-ahead tokens.
- LL parsers are commonly used for languages with deterministic, left-to-right, leftmost derivations, and are suitable for LL(k) grammars.

3. LALR Parser:
- A bottom-up parsing technique that uses a Look-Ahead LR parsing algorithm.
- It employs a parsing table and a state machine to reduce the input tokens to non-terminals, building a parse tree or AST from the bottom up.
- LALR parsers are efficient and can handle a broad class of grammars, including many LR(1) grammars.

4. LR Parser:
- A bottom-up parsing technique that employs LR parsing algorithms.
- LR parsers work by constructing a parse table and using a state machine to shift tokens or reduce them to non-terminals.
- LR parsers can handle a larger class of grammars than LL parsers, including left-recursive and ambiguous grammars.
- Variants of LR parsers include SLR (Simple LR), LALR (Look-Ahead LR), and LR(1) parsers.

5. Earley Parser:
- A chart parsing algorithm that uses dynamic programming to analyze any context-free grammar.
- Earley parsers employ a chart data structure to track and combine partial parse results.
- They can handle a wide range of grammars, including those with ambiguity and left recursion, but may have higher time complexity.

6. CYK Parser (Cocke-Younger-Kasami):
- A parsing algorithm based on dynamic programming and tabulation.
- CYK parsers use a bottom-up approach and work with grammars in Chomsky Normal Form (CNF) or similar forms.
- They build a parse table and apply tabulation techniques to efficiently analyze the input.

Each type of parser has its strengths and weaknesses, and the choice of parser depends on various factors such as the grammar's characteristics, efficiency requirements, and error handling capabilities. Compiler designers and language implementers select an appropriate parser based on the specific requirements of the language being processed.

left recursion

Left recursion is a characteristic of a context-free grammar in which a non-terminal A can directly or indirectly produce a production rule that starts with A itself. In other words, a grammar has left recursion if there is a production rule of the form A -> Aα, where α represents a string of terminals and non-terminals.

Left recursion can create challenges for parsers because it can result in infinite recursion and ambiguity. When left recursion is present, a naive recursive descent parser can go into an infinite loop, continuously expanding the same non-terminal without making any progress in the parsing process.

To handle left recursion, various techniques can be employed:

1. Elimination by Left Factoring:
- Left factoring is a technique that involves rewriting the production rules to eliminate direct left recursion.

- It involves creating new non-terminals and rearranging the production rules to ensure that no rule has direct left recursion.
- For example, given a production A -> Aα | β, left factoring can transform it into A -> βA', A' -> αA' | ε, where ε represents an empty string.

2. Elimination by Left Recursion Removal:

- Another approach is to eliminate left recursion by rewriting the production rules to indirectly reference the non-terminal, rather than directly.
- This technique involves creating new non-terminals and rewriting the rules in a way that avoids left recursion.
- For example, given a production A -> Aα | β, left recursion removal can transform it into A -> βA', A' -> αA' | ε.

By applying these techniques, left recursion can be eliminated, allowing parsers to correctly handle the grammar and avoid issues like infinite recursion and ambiguity. However, it's important to note that left recursion removal may introduce additional complexity and may impact the readability and understanding of the grammar.

Top-down parser

A top-down parser is a type of parsing technique that starts from the top of the parse tree and works its way down, attempting to construct a parse tree or abstract syntax tree (AST) for a given input sequence. It starts with the top-level non-terminal of the grammar and recursively expands it to match the input tokens, following the grammar rules in a left-to-right manner.

Here are some key characteristics of top-down parsers:

1. Recursive Descent Approach:

- Top-down parsers typically use a recursive descent approach, where each non-terminal in the grammar is associated with a parsing function or subroutine.
- The parsing function for a non-terminal is responsible for matching and expanding the non-terminal based on the available input tokens.
- Recursive descent parsers closely mirror the structure of the grammar rules, with each parsing function corresponding to a non-terminal and invoking other parsing functions to handle sub-expressions.

2. Leftmost Derivation:

- Top-down parsers construct a parse tree or AST by following a leftmost derivation, where the leftmost non-terminal is chosen for expansion at each step.
- This means that the parser selects the leftmost non-terminal in the current parse tree node and tries to match it with the available input tokens.

3. LL Parsing:

- Top-down parsers are often referred to as LL parsers, where LL stands for "left-to-right, leftmost derivation."
- The "left-to-right" indicates that the parser scans the input from left to right, matching the tokens in the same order as they appear in the input.
- The "leftmost derivation" signifies that the parser chooses the leftmost non-terminal in each expansion step.

4. Predictive Parsing:

- Top-down parsers can be predictive if they determine the next production rule to apply based solely on the current non-terminal and the current input token (lookahead).
- Predictive parsing allows for efficient and deterministic parsing without requiring backtracking or multiple possibilities for the same non-terminal.

Top-down parsers, specifically recursive descent parsers, are relatively straightforward to implement and understand, as they closely mirror the grammar rules. However, they may face challenges with left recursion and ambiguous grammars, requiring techniques like left factoring or left recursion removal to handle these situations.

First() and Follow()

In the context of parsing and constructing a predictive parsing table, the concepts of First() and Follow() sets are important. These sets assist in determining the possible terminals that can occur as the first and subsequent symbols of a production rule in a given grammar. Let's explore each of these sets:

First() Set:

The First() set represents the set of terminals that can be the first symbol of a string derived from a non-terminal in a grammar. It is used to predict which production rule to apply during parsing. The First() set of a non-terminal A is denoted as First(A). The process of finding the First() set involves considering the following scenarios:

1. If A is a terminal symbol, then First(A) contains only A.
2. If A is a non-terminal symbol and there is a production rule A -> αβ, where α is a sequence of terminals and/or non-terminals, then First(A) contains First(α) (excluding ε, the empty string).
3. If A is a non-terminal symbol and there is a production rule A -> ε, then First(A) contains ε.

By considering these scenarios and recursively applying the rules, the First() set can be calculated for each non-terminal in the grammar.

Follow() Set:
The Follow() set represents the set of terminals that can occur immediately after a non-terminal in a grammar. It helps determine the valid symbols that can follow a non-terminal during parsing. The Follow() set of a non-terminal A is denoted as Follow(A). The process of finding the Follow() set involves considering the following scenarios:

a. The Follow() set of the start symbol of the grammar usually contains the end-of-input marker, denoted as $.
b. If there is a production rule A -> αBβ, where B is a non-terminal symbol, then the terminals in First(β) (excluding ε) are included in Follow(B).
c. If there is a production rule A -> αB or A -> αBβ, where First(β) contains ε, then Follow(A) is included in Follow(B).

By considering these scenarios and iteratively applying the rules, the Follow() set can be calculated for each non-terminal in the grammar.

The First() and Follow() sets are typically used in predictive parsing algorithms, such as LL(1) parsers, to construct a parsing table. The parsing table uses the First() set to predict the production rule based on the next input symbol, and the Follow() set to handle non-terminals during parsing decisions. **These** sets play a vital role in determining the validity of the input and guiding the parsing process, ensuring the correct construction of parse trees or abstract syntax trees according to the grammar rules.

LL(1) Grammars

LL(1) grammars refer to a class of grammars that can be parsed by a predictive LL(1) parser. The "LL" stands for "left-to-right, leftmost derivation," and the "1" indicates that the parser looks ahead at most one token to make parsing decisions. An LL(1) grammar must satisfy certain properties to be suitable for parsing using this technique.

Here are the characteristics and requirements of an LL(1) grammar:

1. Deterministic Productions:
- The grammar should have deterministic (unambiguous) production rules, meaning that for any non-terminal and lookahead token, there should be at most one possible production rule to apply.
- Ambiguities, such as left recursion or common prefixes in alternative productions, are not allowed in an LL(1) grammar.

2. Left Factoring:
- To handle alternatives with common prefixes, an LL(1) grammar often requires left factoring, which involves rewriting the grammar rules to eliminate conflicts.
- Left factoring ensures that the parser can determine the appropriate production rule based on the lookahead token without requiring additional lookahead.

3. First() and Follow() Sets:
- An LL(1) grammar should have unambiguous First() and Follow() sets for all non-terminals.
- The First() set should not contain any common prefixes among the alternatives of a non-terminal.
- The Follow() set should not intersect with the First() set of any alternative of the same non-terminal.

4. No ε-Productions:
- An LL(1) grammar should not have ε-productions, which are production rules that derive the empty string (ε).
- Handling ε-productions in LL(1) parsing can lead to ambiguity or conflicts in the parsing table.

By satisfying these properties, an LL(1) grammar can be effectively parsed using a predictive LL(1) parsing algorithm, such as recursive descent or table-driven parsing. The LL(1) parsing algorithm constructs a parsing table based on the First() and Follow() sets of the grammar's non-terminals, which guides the parsing decisions without requiring backtracking.

LL(1) grammars are commonly used for top-down parsing of programming languages and other formal languages. They offer simplicity and efficiency in parsing, making them suitable for languages with deterministic, left-to-right, leftmost derivations.

Non- Recursive predictive parsing

Non-recursive predictive parsing, also known as table-driven predictive parsing, is an alternative approach to recursive descent parsing for LL(1) grammars. While recursive descent parsing uses recursive function calls to match non-terminals with production rules, table-driven parsing employs a parsing table that directly maps non-terminals and lookahead tokens to production rules.

Here's an overview of how non-recursive predictive parsing works:

1. Constructing the Parsing Table:
- The first step is to construct a parsing table, also known as a predictive parsing table or LL(1) parsing table.
- The parsing table is typically a two-dimensional data structure that represents the grammar's non-terminals and terminals.
- Each entry in the parsing table specifies the production rule to be applied when a non-terminal and lookahead token combination is encountered during parsing.

2. Populating the Parsing Table:
- To populate the parsing table, the First() and Follow() sets are calculated for each non-terminal in the grammar.

- For each non-terminal A and each terminal a in First(A), add the production A -> α to the parsing table entry [A, a], where α is the production rule.
- If ε is in First(A), for each terminal b in Follow(A), add the production A -> ε to the parsing table entry [A, b].

3. Parsing Process:

- During the parsing process, an input string of tokens is matched against the parsing table.
- A stack is used to keep track of the non-terminals and terminals that have been processed.
- Initially, the start symbol of the grammar is pushed onto the stack, and the first token of the input is read.
- While the stack is not empty, the top of the stack (non-terminal) and the current input token are used to determine the production rule to apply.
- The production rule is retrieved from the parsing table based on the stack's top non-terminal and the current input token.
- The non-terminal is replaced with the right-hand side of the production rule, and the stack and input are updated accordingly.
- The process continues until the input is fully parsed and the stack becomes empty, indicating a successful parsing.

Non-recursive predictive parsing eliminates the need for explicit recursive function calls and stack management, making it more efficient and less prone to stack overflow issues compared to recursive descent parsing. The parsing table provides a direct mapping between non-terminals, terminals, and production rules, simplifying the parsing process.

Bottom-up Parsers

Bottom-up parsers are a class of parsing algorithms that build parse trees or abstract syntax trees (ASTs) from the input string by starting from the leaves (tokens) and working their way up to the root. These parsers attempt to find a valid rightmost derivation of the input string, often using a stack-based approach.

Here are some important types of bottom-up parsers:

1. Shift-Reduce Parsers:

- Shift-reduce parsers work by shifting input tokens onto a stack until a right-hand side of a production rule is found. At this point, a reduction operation is performed, replacing the right-hand side with its corresponding non-terminal symbol.
- The shift-reduce process continues until the entire input string is reduced to the start symbol.
- LR (Left-to-right, Rightmost derivation) parsers are a common type of shift-reduce parsers. Variants of LR parsers include SLR (Simple LR), LALR (Look-Ahead LR), and LR(1) parsers, which differ in their lookahead capabilities and the size of their parsing tables.

2. LR Parser:

- LR parsers are a class of bottom-up shift-reduce parsers.
- They use deterministic finite automata and a parsing table to guide the shift and reduce operations.
- LR parsers can handle a wide range of grammars, including left-recursive and ambiguous grammars, and are more powerful than LL parsers.
- LR parsing algorithms, such as LR(0), SLR(1), LALR(1), and LR(1), differ in their capabilities and the amount of lookahead they use.

3. Operator Precedence Parser:

- Operator precedence parsers are bottom-up parsers that use operator precedence and associativity rules to parse expressions.
- These parsers operate by scanning the input and comparing the precedence of operators to determine their grouping and reduce them into higher-level expressions.
-

Bottom-up parsing has several advantages:

- It can handle a broader class of grammars, including left-recursive and ambiguous grammars.
- Bottom-up parsers are more powerful and flexible than top-down parsers.
- They allow for efficient error recovery by using lookahead and backtracking.

Shift Reduce Parser

A shift-reduce parser is a type of bottom-up parser that operates by shifting input tokens onto a stack and then reducing them based on predefined production rules. It employs a shift operation to read input symbols and push them onto the stack, and a reduce operation to apply production rules and replace a group of symbols on the stack with their corresponding non-terminal symbol. This process continues until the entire input is reduced to the start symbol, indicating a successful parse.

Here's an overview of how a shift-reduce parser works:

1. Parsing Table:

- The shift-reduce parser uses a parsing table, which is typically generated from the grammar and provides instructions for the shift and reduce operations.
- The parsing table specifies actions based on the current state of the parser and the lookahead token.
- The actions in the parsing table can be either shift, reduce, or accept (indicating successful parsing).

2. Stack and Input:
- The parser maintains a stack to keep track of the symbols it has encountered during parsing.
- Initially, the start symbol is pushed onto the stack, and the input string of tokens is prepared for parsing.

3. Parsing Process:
- While the stack is not empty and the input is not fully processed:
- Look at the current state of the parser (top of the stack) and the lookahead token (next input symbol).
- Consult the parsing table to determine the appropriate action to take based on the current state and lookahead token.
- If the action is a shift operation, shift the lookahead token onto the stack and move to the next state.
- If the action is a reduce operation, apply the production rule specified in the parsing table by popping the appropriate number of symbols from the stack and replacing them with their corresponding non-terminal.
- If the action is an accept operation, the input has been successfully parsed, and the parsing process terminates.

4. Error Handling:
- The parsing table may also specify error actions to handle situations where the current state and lookahead token do not match any valid action.
- Common error actions include reporting a syntax error and attempting error recovery strategies, such as discarding tokens or inserting missing tokens to continue parsing.

Shift-reduce parsers, such as LR parsers, are powerful and can handle a wide range of grammars, including left-recursive and ambiguous grammars. However, constructing and managing the parsing table can be more complex compared to top-down parsing techniques like LL parsers. Various LR parser variants, such as SLR(1), LALR(1), and LR(1), differ in their lookahead capabilities and table sizes, offering different trade-offs between efficiency and expressive power.

LR parsers

LR parsers are a class of bottom-up shift-reduce parsers that can handle a wide range of context-free grammars. The "LR" in LR parser stands for "Left-to-right, Rightmost derivation," which reflects the parsing strategy employed by these parsers.
LR parsers operate by reading the input from left to right and constructing a rightmost derivation of the input string. They utilize a deterministic finite automaton called a parser state machine and a parsing table to guide the shift and reduce operations during parsing.

Here are some key aspects of LR parsers:

1. Deterministic Finite Automaton:
- LR parsers use a deterministic finite automaton, also known as the LR state machine or LR automaton, to keep track of the parser's state during parsing.
- The state machine transitions between states based on the symbols it encounters from the input.

2. LR Parsing Table:
- The parsing table is a data structure used by LR parsers to determine the actions to be taken based on the current state and lookahead token.
- The parsing table specifies whether to shift a token onto the stack or perform a reduce operation based on the current state and lookahead token.
- The parsing table is typically generated from the grammar using LR parsing algorithms such as LR(0), SLR(1), LALR(1), or LR(1).

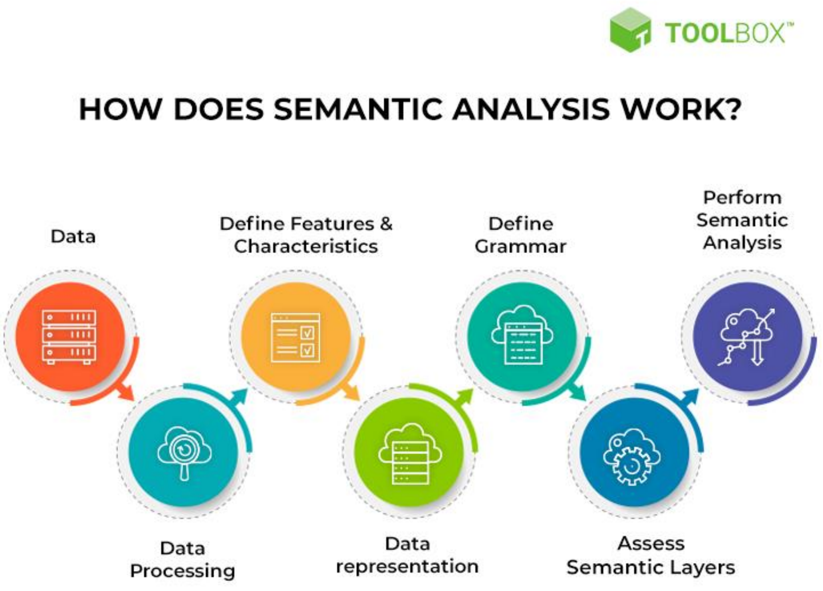
3. LR Parsing Algorithms:
- LR parsing algorithms determine the properties and construction of the parsing table.
- LR(0) parsing is the simplest form, where the parsing table is constructed solely based on the states and transitions of the LR state machine.
- SLR(1) parsing and LALR(1) parsing enhance the LR(0) algorithm by incorporating additional lookahead information to resolve conflicts and reduce the size of the parsing table.
- LR(1) parsing is the most powerful and expressive variant, as it uses a larger amount of lookahead information to construct the parsing table, allowing for a broader range of grammars to be parsed.

4. Bottom-Up Parsing:
- LR parsers follow a bottom-up parsing approach, where they start by shifting input tokens onto the stack until a valid production rule can be reduced.
- The shift operation involves pushing the input token onto the stack and transitioning to a new state in the parser state machine.
- The reduce operation replaces a group of symbols on the stack with their corresponding non-terminal, based on a valid production rule.

LR parsers offer several advantages:
- They can handle a broad class of context-free grammars, including left-recursive and ambiguous grammars.
- They provide efficient and deterministic parsing, as they use a finite state machine and a parsing table to guide the parsing decisions.
- LR parsers allow for efficient error recovery and reporting, thanks to their built-in error handling mechanisms.

<u>Semantic Analysis:</u>

Semantic analysis is a phase in the compilation process where the meaning and correctness of the source code are analyzed beyond its syntactic structure. It focuses on understanding the semantics or the intended behavior of the program by considering the context, type compatibility, and adherence to language-specific rules and constraints.


TOOLBOX™
HOW DOES SEMANTIC ANALYSIS WORK?
Data
Data Processing
Define Features & Characteristics
Data representation
Define Grammar
Assess Semantic Layers
Perform Semantic Analysis

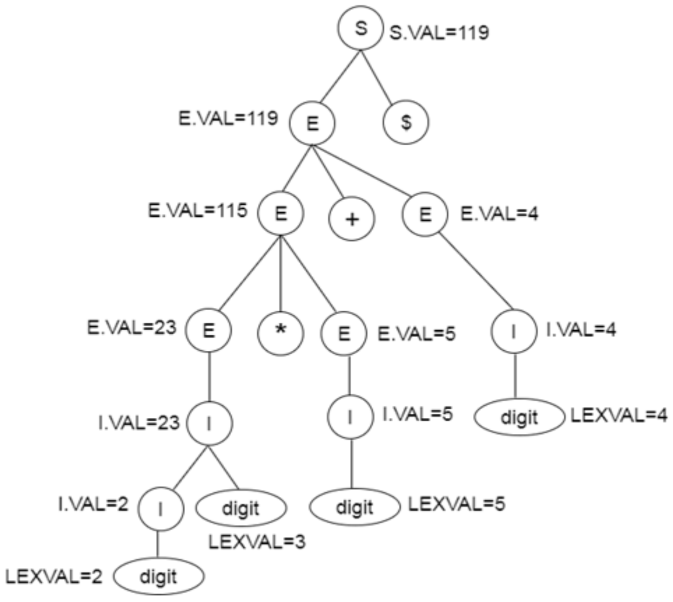
<u>Here are the key aspects of semantic analysis:</u>

1. Type Checking:
- Type checking is a fundamental task in semantic analysis that ensures the compatibility and consistency of data types used in expressions, assignments, function calls, and other operations.
- It verifies that operations are performed on operands of compatible types, and it detects type mismatches or inconsistencies.
2. Symbol Table Management:
- Symbol tables are data structures used during semantic analysis to store information about variables, functions, classes, and other program elements.
- Symbol table management involves building and maintaining the symbol table, which records the attributes and characteristics of symbols encountered in the source code, such as their data types, scopes, and declaration locations.
- The symbol table is used for identifier resolution, checking for undeclared or conflicting symbols, and enforcing scoping rules.
3. Scope Analysis:
- Scope analysis determines the visibility and accessibility of variables, functions, and other program entities in different parts of the code.
- It enforces scoping rules, such as block-level scoping or lexical scoping, and checks for correct usage of variables within their declared scopes.
4. Error Detection and Reporting:
- Semantic analysis identifies and reports various semantic errors that cannot be caught during lexical and syntax analysis.
- Examples of semantic errors include type mismatches, undeclared variables, redeclaration of symbols, incorrect use of functions or operators, and violation of language-specific rules.
5. Intermediate Code Generation:
- In some compilers, semantic analysis also involves generating intermediate code representations, such as Abstract Syntax Trees (AST) or Three-Address Code (TAC).
- Intermediate code generation simplifies further optimizations and the translation of the high-level source code into low-level machine code or an executable format.

Semantic analysis plays a crucial role in ensuring the correctness, safety, and meaningful interpretation of the source code. By detecting semantic errors and enforcing language-specific rules, it helps in producing reliable and efficient executable code. Semantic analysis is typically performed after the lexical and syntactic analysis phases, and it is followed by optimization and code generation stages in the compilation process.

Syntax Directed Translation

Syntax-directed translation is a method used in compiler design to associate semantic actions with productions in a grammar. It combines the parsing and translation phases by embedding translation code within the production rules of a grammar. Each production rule is associated with specific actions that are executed during parsing to perform the desired translation or semantic analysis.



Here's how syntax-directed translation works:

1. Extended Grammar:
- The grammar is extended with semantic rules or attributes associated with each production rule.
- These rules or attributes capture additional information or computations required during the translation or semantic analysis phase.
2. Inherited and Synthesized Attributes:
- Attributes are used to pass information between non-terminals and terminals during parsing.
- Inherited attributes are passed from parent non-terminals to child non-terminals in the parse tree.
- Synthesized attributes are computed at each non-terminal and propagate their values up to the parent non-terminals.
3. Semantic Actions:
- Semantic actions are embedded within the production rules of the grammar.
- These actions are executed during parsing and perform specific translations, computations, or semantic checks associated with the corresponding production.
- Semantic actions can manipulate attribute values, update symbol tables, generate intermediate code, or perform other tasks specific to the programming language or compiler requirements.
4. Parse Tree Construction:
- As the parser processes the input according to the grammar and executes the semantic actions, a parse tree or an abstract syntax tree (AST) is constructed.
- The parse tree or AST captures the syntactic structure of the input and also incorporates the translated or analyzed information obtained through the semantic actions.
- Syntax-directed translation allows for the integration of translation and semantic analysis tasks directly into the parsing process. It facilitates the generation of intermediate representations, code optimization, or target code generation alongside the syntactic analysis.

This approach provides several benefits:
- It simplifies the compiler design by combining parsing and translation phases.
- It allows for efficient traversal of the parse tree, enabling immediate translation and analysis as the input is parsed.
- It ensures that translation or semantic analysis is tightly coupled with the grammar rules, improving clarity and maintainability.

L-attributed and S-attributed definitions

M-attributed and S-attributed are two types of syntax-directed translation schemes that specify the flow of attribute values within a grammar. These schemes determine the order in which attribute values are computed and propagated during parsing and translation.

1. L-attributed Definitions:
- L-attributed definitions are a type of syntax-directed translation scheme where attribute values can be determined from left to right in a production rule without requiring any lookahead.
- In other words, the values of attributes in L-attributed definitions can be computed solely based on inherited attributes and attributes of symbols on the left-hand side of the production rule.
- The attributes in an L-attributed definition are typically associated with non-terminals and are computed using semantic rules embedded within the grammar.
- L-attributed definitions are used in grammars where attribute values can be evaluated in a bottom-up manner, such as in S-attributed definitions.

2. S-attributed Definitions:
- S-attributed definitions are a more general type of syntax-directed translation scheme where attribute values can depend on both inherited attributes and attributes of symbols on the right-hand side of the production rule, including lookahead symbols.
- In S-attributed definitions, the attribute values may need additional information or computations beyond the left-to-right propagation of L-attributed definitions.
- S-attributed definitions allow for more flexibility in specifying and evaluating attribute dependencies within a grammar.
- S-attributed definitions are used when the attribute values cannot be fully determined in a bottom-up manner and require additional context or computations.

Both L-attributed and S-attributed definitions play important roles in syntax-directed translation:
- L-attributed definitions are commonly used in bottom-up parsing techniques, such as LR parsing, where attribute values can be evaluated in a bottom-up fashion based on inherited attributes and attributes of symbols on the left-hand side of the production rule.
- S-attributed definitions are used when more complex dependencies exist between attribute values, including dependencies on symbols on the right-hand side of the production rule or lookahead symbols. S-attributed definitions are often used in more expressive parsing techniques like generalized LR (GLR) parsing.

Aspect	L-attributed Definitions	S-attributed Definitions
Attribute Dependencies	Depends on inherited attributes and symbols on the left-hand side of the production rule	Can depend on inherited attributes, symbols on both sides of the production rule (including right-hand side), and lookahead symbols
Evaluation Order	Can be evaluated from left to right without requiring lookahead	May require additional context or computations beyond left-to-right propagation
Bottom-Up Evaluation	Suitable for bottom-up parsing techniques like LR parsing	Can be used in more expressive parsing techniques like generalized LR (GLR) parsing
Attribute Propagation	Values are computed and propagated bottom-up, relying on inherited attributes and symbols on the left-hand side	Values can depend on inherited attributes, symbols on both sides, and lookahead symbols
Complexity	Simpler and more restricted, suitable for many practical cases	More flexible and capable of handling a broader range of attribute dependencies
Common Usage	Commonly used in traditional LR parsing and simpler grammars	Used in more complex grammars with additional dependencies or computations