Module 3: Syntax Analysis (Parsing)

This module is about the compiler's crucial "grammar check" phase: Syntax Analysis, also known as **Parsing**. After the Lexical Analyzer has broken the raw source code into meaningful "words" (**tokens**), the parser steps in to ensure these words are arranged according to the language's rules, forming grammatically correct "sentences" and "paragraphs." We'll explore the formal definitions of language structure, how parsers systematically verify this structure, and the tools that automate this complex task.

---

1. Context-Free Grammars (CFG) and Language Structure

Think of a **Context-Free Grammar (CFG)** as the definitive rulebook or blueprint for a programming language's syntax. It's a formal, mathematical way to describe all the possible legal sequences of words (tokens) that can form valid programs in that language. Without a precise grammar, a compiler wouldn't know how to interpret your code.

A CFG is precisely defined by four components:

- **V (Variables / Non-terminals)**: These are abstract, conceptual symbols that represent structures or constructs within the language. They are "non-terminal" because they are not the final "words" of the program but rather categories that can be broken down further (e.g., `Statement`, `Expression`, `Program`).
- **T (Terminals)**: These are the actual, tangible "words" or tokens that the lexical analyzer produces. They are "terminal" because they cannot be broken down further within the grammar rules; they are the basic building blocks (e.g., keywords like `if`, operators like `+`, punctuation like `;`, literal values like `NUM`, `ID`).
- **P (Productions / Production Rules)**: These are the core rules that specify how non-terminals can be replaced by sequences of other non-terminals and/or terminals. Each rule is like a recipe that tells you how to construct a larger grammatical unit from smaller ones.
    - **Format**: `Non-terminal -> Sequence_of_Symbols`
    - **Example**: `Statement -> if ( Expression ) Statement else Statement`
- **S (Start Symbol)**: This is a special non-terminal that represents the highest-level grammatical category in the language. It's the ultimate goal of the parsing process. A successful parse means that the entire input program can be derived from this start symbol (e.g., `Program`).

Why CFGs are Important:

- **Formal Specification**: They provide an unambiguous way to define the syntax of a language.
- **Automatic Parser Generation**: They are the input for tools that automatically build the parser component.
- **Error Detection**: If a program's structure doesn't conform to the CFG, the parser can detect and report syntax errors.

Example CFG (for a tiny arithmetic calculator, expanded):

- **V** = {Program, Statement, Expression, Term, Factor, IDList}
- **T** = { ID, NUM, +, -, *, /, (,), =, ;, var }
- **S** = Program
- **P**:
    1. Program -> Statement Program
    2. Program -> ε (epsilon denotes an empty string)
    3. Statement -> var IDList ;
    4. Statement -> ID = Expression ;
    5. IDList -> ID
    6. IDList -> ID , IDList
    7. Expression -> Expression + Term
    8. Expression -> Expression - Term
    9. Expression -> Term
    10. Term -> Term * Factor
    11. Term -> Term / Factor
    12. Term -> Factor
    13. Factor -> (Expression)
    14. Factor -> ID
    15. Factor -> NUM

---

2. The Parsing Process: Concepts, Trees, and Derivations

Parsing is the compiler's "syntax police." Its job is to take the stream of tokens from the lexical analyzer and determine if they form a grammatically valid program according to the rules defined by the Context-Free Grammar. If the arrangement of tokens makes sense structurally, the parser creates a hierarchical representation of the program. If not, it flags a syntax error.

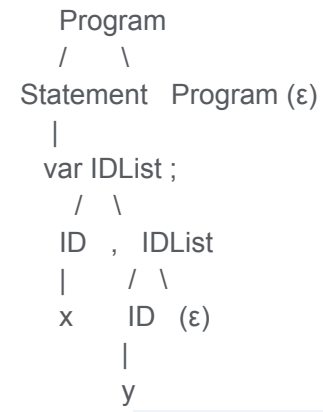
What Parsing Achieves:

- **Syntax Validation**: Ensures that your code follows the structural rules of the language.
- **Structure Representation**: It builds a tree-like structure that captures the relationships between different parts of your code.

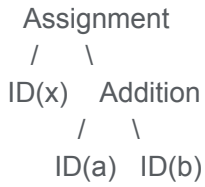
Parse Trees vs. Abstract Syntax Trees (AST)

- **Parse Tree (Concrete Syntax Tree)**:
    - This is a visual, detailed representation of how the input string (sequence of tokens) is derived from the start symbol of the grammar using the production rules.
    - **Characteristics**: Root is the start symbol; internal nodes are non-terminals (corresponding to a production rule application); leaf nodes are terminal symbols (tokens). Reading leaf nodes from left to right gives the original input.

    - **Purpose**: Shows every single step of the derivation process, including intermediate non-terminals used purely for grammatical structure.
    - **Example Parse Tree for `var x , y ;` using our calculator grammar:**



- **Abstract Syntax Tree (AST)**:
    - While a parse tree shows all grammatical details, an AST is a more compact and essential representation of the program's structure. It focuses on the core operations and relationships, stripping away syntactic details that aren't directly relevant to the program's meaning.
    - **Why Abstract?**: It removes "noise" from the parse tree (e.g., omitting non-terminals like `Term` or `Factor` if their sole purpose was to enforce operator precedence). It only keeps nodes that represent a computational or structural meaning.
    - **Purpose**: The AST is the preferred input for later compiler phases like semantic analysis and code generation, as these phases care about the meaning and relationships of operations.
    - **Example AST for `x = a + b ;` (simplified):**



Sentences and Sentential Forms - The Stages of Derivation

- **Derivation**: This is the process of repeatedly applying the production rules of a CFG, starting from the start symbol, to transform one string of symbols into another.

- **Sentential Form**: Any string that can be derived from the start symbol of a grammar. This string can contain a mixture of both non-terminal symbols (abstract categories) and terminal symbols (concrete words). It's an intermediate stage in building a complete program "sentence."
    - **Example**: `Statement Program`, `var IDList ; Program` are sentential forms.
- **Sentence**: A special type of sentential form consisting *only* of terminal symbols. It represents a complete and grammatically valid program (or a segment of one) in the language.
    - **Example**: `var x , y ;` is a sentence.

Leftmost and Rightmost Derivations - Following a Path in the Tree

When a sentential form contains multiple non-terminals, we choose which one to expand next:

- **Leftmost Derivation**: Always chooses the leftmost non-terminal in the current sentential form to replace. This is a common strategy for top-down parsers.
- **Rightmost Derivation (Canonical Derivation)**: Always chooses the rightmost non-terminal in the current sentential form to replace. This strategy is more common for bottom-up parsers.

Both derivations produce the exact same final string and result in the same parse tree, even if the order of rule applications differs.

---

3. Ambiguous Grammars and Resolution

A grammar is considered **ambiguous** if there is at least one sentence (a valid string of terminals) in the language that can be derived in more than one distinct way. This means the sentence has:

- More than one unique parse tree.
- Or, more than one distinct leftmost derivation.
- Or, more than one distinct rightmost derivation.

Why Ambiguity is a Problem: In programming languages, ambiguity leads to uncertainty about the program's intended meaning. If `A - B * C` could be interpreted as `(A - B) * C` or `A - (B * C)`, the compiled code would behave differently, leading to unpredictable errors. A compiler must have a single, definitive way to parse every valid program.

Classic Example: Arithmetic Expressions without Precedence/Associativity Rules

Consider this simple, ambiguous grammar:

E -> E + E

E -> E * E

E -> ID

The input `a + b * c` can result in two fundamentally different parse trees, implying different orders of operation:

- **Parse Tree 1 (implies `(a + b) * c`):**

```
    E
    /|\
   E + E
  /  /|\
 ID  E * E
 |  |   |
 a  ID  ID
    |   |
    b   c
```

-
-
- **Parse Tree 2 (implies `a + (b * c)`):**

```
    E
    /|\
   E + E
  /  / \
 ID  E   E
 |  /|\  |
 a E * E ID
   |   |
   b   c
```

-
-

Since a single input string results in two different parse trees, this grammar is ambiguous.

Resolving Ambiguity: Compiler designers use two primary mechanisms:

1. **Precedence Rules**: Define the order in which operators are evaluated (e.g., `*` and `/` have higher precedence than `+` and `-`).
    - **Implementation in Grammar**: Rewrite the grammar by introducing new non-terminals to create a hierarchy where higher-precedence operations are "lower down" (closer to terminals) in the parse tree.
    - Example (for * having higher precedence than +):
      Expression -> Expression + Term
      Expression -> Term
      Term -> Term * Factor
      Term -> Factor
      Factor -> ID (or (Expression))
2. **Associativity Rules**: Define how operators of the same precedence are grouped when they appear sequentially.

    - **Left Associativity**: `a - b - c` is `(a - b) - c`. Implemented using **left-recursive** production rules (e.g., `Expression -> Expression + Term`).
    - **Right Associativity**: `a = b = c` is `a = (b = c)`. Implemented using **right-recursive** production rules (e.g., `Assignment -> ID = Assignment`).

By carefully applying these rules and rewriting the grammar, language designers ensure that every syntactically correct program has only one unambiguous interpretation.

---

4. Parsing Strategies: Top-Down vs. Bottom-Up

The two fundamental strategies for parsing a program relate to how they build the parse tree.

Top-Down Parsing (Predictive Parsing)

- **Approach**: "Building from the Blueprint Down." Starts at the start symbol (the root of the parse tree) and tries to expand it downwards to match the input tokens (the leaves).
- **How it Works**: The parser tries to predict which production rule for a non-terminal should be used next to match the incoming input tokens. It essentially tries to construct a leftmost derivation.
- **Characteristics**:
    - Easier to implement manually for simpler grammars.
    - Requires the grammar to be free of left recursion and often left factoring.
    - Less powerful than bottom-up parsers (can't handle as wide a range of grammars).
- **Common Techniques**: Recursive Descent Parsing, Predictive Parsing (LL(1)).

Bottom-Up Parsing (Shift-Reduce Parsing)

- **Approach**: "Assembling from the Pieces Up." Starts with the input tokens (the leaves of the parse tree) and attempts to combine them (reduce them) into higher-level grammatical constructs, eventually reducing everything to the start symbol (the root).
- **How it Works**: The parser scans the input, shifting tokens onto a stack. When the top of the stack contains a sequence of symbols that matches the right-hand side of a production rule, it "reduces" that sequence to the non-terminal on the left-hand side of the rule. This effectively builds the parse tree from the leaves upwards towards the root, constructing a rightmost derivation in reverse.
- **Characteristics**:
    - More powerful; can handle a larger class of grammars than top-down parsers.
    - Often more complex to implement manually but well-suited for automatic generation by tools.
    - No issues with left recursion.
- **Common Techniques**: Shift-Reduce Parsing, LR Parsers (LR(0), SLR(1), LALR(1), LR(1)).

---

5. Bottom-Up Parsing in Detail: Shift-Reduce and SLR

This section dives into **Shift-Reduce parsing**, the core idea behind powerful bottom-up parsers like the LR family.

Introduction to Shift-Reduce Parsing

Shift-Reduce parsing is a strategy that operates by trying to find the "handle" in the parser's stack. A handle is a substring on the stack that matches the right-hand side of a grammar production and can be "reduced" to its corresponding non-terminal.

The Parser's Tools:

- **Input Buffer**: Where the raw stream of tokens from the lexical analyzer waits.
- **Stack**: The parser's primary working area, storing a sequence of grammar symbols.
- **Parsing Table**: Pre-computed from the grammar, it tells the parser what to do (shift, reduce, accept, or error) based on the current state (top of stack) and the next input token (the "lookahead" symbol).

The Core Actions: The parser continuously performs one of these actions:

- **Shift**: Takes the next incoming token from the input buffer and pushes it onto the stack.
- **Reduce**: When symbols on the top of the stack match the entire right-hand side (beta) of a production rule (Atobeta), the parser pops these matched symbols and pushes the non-terminal A onto the stack. This signifies that a complete grammatical construct has been recognized.
- **Accept**: If the stack contains only the start symbol (S′) and the input buffer is empty, the entire program has been successfully parsed.
- **Error**: If the parser cannot perform a valid action, a syntax error is reported.

Example Walkthrough: Parsing a + b with Shift-Reduce (Assume simplified grammar: E -> E + E | ID)

Initial state: Stack: $ | Input: a + b $ (where $ marks end of input)

Stack	Input	Action (determined by Parsing Table)	Explanation
$	a + b $	Shift a	Push a onto stack.
$a	+ b $	Reduce E -> ID	a is an ID. Reduce ID to E. Pop a, push E.

$E	+ b $	Shift +	Push + onto stack.
$E+	b $	Shift b	Push b onto stack.
$E+b	$	Reduce E -> ID	b is an ID. Reduce ID to E. Pop b, push E.
$E+E	$	Reduce E -> E + E	E + E is on top of stack. Reduce to E. Pop E,+,E, push E.
$E	$	Accept	Stack has start symbol, input empty. Success!

Viable Prefixes and Valid Items - Guiding the Parser's Decisions

To build robust bottom-up parsers, we use "items" to describe the parser's progress.

- **Viable Prefix**: Any prefix of a rightmost sentential form that can exist on the stack during a shift-reduce parse. The parser's stack always holds a viable prefix.
    - **Example**: $, $a, $E, $E+, $E+b are all viable prefixes in the example above.
- **Item (LR(0) Item)**: A production rule with a "dot" (.) placed somewhere in its right-hand side. The dot indicates how much of the right-hand side has been recognized so far.
    - **Format**: A -> α . β (where α is matched, β is expected)
    - **Example Items (for E -> E + E and E -> ID)**:
        - E -> . E + E (expecting E + E)
        - E -> E . + E (matched E, expecting + E)
        - E -> E + E . (matched E + E, ready to reduce to E)
        - E -> ID . (matched ID, ready to reduce to E)
    - **Role in Parsing**: LR parsers build "states," where each state is a collection of items, representing a snapshot of all possible production rules the parser could be trying to recognize.

Constructing LR(0) Sets of Items - Defining Parser States

To create an LR parser, we define all its possible "states" using "sets of LR(0) items."

1. **Augmented Grammar**: Add a new start production S′toS to the original grammar. This ensures a single, clear reduction (S′toS.) signals successful parsing.

2. **CLOSURE Operation**: Expands a set of items. If an item Atoalpha.Bbeta (expecting non-terminal B) is in a set, then for every production Btogamma, add Bto.gamma to the set. Repeat until no new items can be added.
3. **GOTO Operation**: Determines the next state after recognizing a grammar symbol X. For a set of items I and symbol X, it finds all items in I where the dot is before X (Atoalpha.Xbeta), moves the dot past X (AtoalphaX.beta), and then takes the CLOSURE of this new set of items.
4. **Building the Canonical Collection of LR(0) Items**: This algorithm generates all unique states. It starts with an initial state I_0=CLOSURE(S′to.S). Then, for each state I and every grammar symbol X that appears after a dot in I, it computes J=GOTO(I,X). If J is new, it's added to the collection and processed.

The resulting collection of states forms the basis for constructing the LR parsing tables.

Constructing SLR Parsing Tables (Simple LR)

SLR (Simple LR) parsing leverages the LR(0) sets of items but adds the **FOLLOW set** to make reduce decisions.

SLR Parsing Table Structure:

- **ACTION Table**: `ACTION[State_i, Terminal_a]` can be:
  - `shift j`: Push `a` and transition to state `j`.
  - `reduce A -> β`: Pop `|β|` symbols, push `A`, and use the GOTO table.
  - `accept`: Input successfully parsed.
  - `error`: Syntax error.
- **GOTO Table**: `GOTO[State_i, NonTerminal_A] = State_j`: The state to transition to after reducing to `NonTerminal_A` from `State_i`.

Rules for Constructing SLR Parsing Table Entries:

- **Shift Actions**: For Atoalpha.abeta in state I_i where a is a terminal, and GOTO(I_i,a) is I_j, set `ACTION[i, a] = shift j`.
- **Reduce Actions**: For Atoalpha. in state I_i (entire right-hand side matched), then for every terminal `b` in FOLLOW(A), set `ACTION[i, b] = reduce A -> α`. The `FOLLOW(A)` set is crucial, ensuring a reduction is performed only if `b` can legally follow `A`.
- **Accept Action**: If S′toS. is in state I_i, set `ACTION[i, $] = accept`.
- **GOTO Actions**: If GOTO(I_i,A) is I_j (where A is a non-terminal), set `GOTO[i, A] = j`.

SLR Conflicts: A grammar is SLR(1) if its SLR parsing table contains no conflicts. Conflicts arise if a cell in the ACTION table has multiple entries:

- **Shift/Reduce Conflict**: A state implies both a shift on a terminal `a` and a reduce action.
- Reduce/Reduce Conflict: A state implies reductions by two different rules for the same lookahead terminal.
  These conflicts indicate that the grammar is not suitable for SLR(1) parsing.
