1. Introduction

The parser is the second phase of the SPL compiler. Its primary role is to take the sequential stream of tokens generated by the lexer and validate that they conform to the language's grammatical rules.

This phase is responsible for identifying the program's hierarchical structure. If the token stream is valid, the parser produces an **Abstract Syntax Tree (AST)**. This AST is the crucial data structure that will be consumed during the next phase the next phase: **Semantic Analysis (Tasks 4-6)**.

If the token stream is *not* valid (i.e., a syntax error is present), the parser will halt and report a `ParseError` with location information.

---

2. Parser Design

This is a **Recursive Descent Parser**. This approach is a top-down parsing strategy that uses a set of mutually recursive functions to process the input.

- **Grammar-Based:** The parser is a direct implementation of the **LL(1) compatible grammar** defined in `LL(1).pdf`.
- **Structure:** The `Parser` class in `parser_spl.py` contains one parsing method for almost every non-terminal symbol in the grammar (e.g., `_parse_spl_prog()`, `_parse_algo()`, `_parse_term()`).
- **Predictive (LL(1)):** The parser uses a single token of **lookahead** (the `current_tok`) to deterministically decide which production rule to follow at any point, as defined by the LL(1) grammar.

---

3. Expected Input

The parser's input is the list (or iterator) of `Token` objects generated by the `Lexer`. Each `Token` object (defined in `lexer.py`) has at least:

- **`type`** **(str):** A string representing the token's category (e.g., `GLOB`, `ID`, `NUMBER`, `LBRACE`, `PLUS_WORD`).
- **`value`** **(Optional[object]):** The actual data for `ID`, `NUMBER`, and `STRING` tokens (e.g., `'myvar'`, `42`, `"hello"`).

- **`line` (int):** The line number for error reporting.
- **`column` (int):** The column number for error reporting.

The parser expects the token stream to end with an `EOF` token, which it must successfully match to signify a complete parse.

---

4. How Processing Works

The parser starts by calling the function for the grammar's start symbol, `_parse_spl_prog()`, and proceeds recursively.

1. **Token Consumption:** The parser manages its position in the token stream. The `_match()` method is used to consume an expected terminal token (e.g., `_match('GLOB', 'glob')`). If the current token doesn't match the expectation, a `ParseError` is raised.
2. **Recursive Calls:** When a grammar rule requires a non-terminal (e.g., `MAINPROG` requires `ALGO`), the parser makes a recursive call to that non-terminal's function (e.g., `_parse_algo()`).
3. **LL(1) Decisions:** In functions where the grammar has multiple options (e.g., `INSTR` can be `halt`, `print`, `if`, etc.), the parser checks the `type` of the `current_tok` (the lookahead) to decide which path to take. For example, in `_parse_instr()`, if `self.current_tok.type == 'IF'`, it calls `_parse_branch()`.
4. **AST Construction:** Each parsing function is responsible for building and returning an **AST Node** (defined in `ast_nodes.py`) corresponding to the structure it has just parsed. It does this by collecting the results (terminals or other AST nodes) from its `_match()` and recursive calls.

---

5. Expected Output (The Abstract Syntax Tree)

The `parse()` method returns a single, complete **`ProgramNode`** object, which is the root of the AST. This tree is the **sole input for the Semantic Analysis phase**.

The next phase requires the creation of a **tree-crawling algorithm** to visit the nodes of this AST. All node classes are defined in `ast_nodes.py`.

Key AST Node Structures (from `ast_nodes.py`):

- **`ProgramNode(globals, procs, funcs, main)`**: The root.
  - `globals`: A `VariableDeclsNode`.

    - `procs`: A `ProcDefsNode`.
    - `funcs`: A `FuncDefsNode`.
    - `main`: A `MainProgNode`.
- **`VariableDeclsNode(variables)`**: Contains a `list` of `VarNode` objects.
- **`VarNode(name)`**: A leaf node holding the string `name` of an identifier.
- **`ProcedureDefNode(name, params, body)`** / **`FunctionDefNode(name, params, body, return_atom)`**:
    - `name`: A `VarNode`.
    - `params`: A `Max3Node`.
    - `body`: A `BodyNode`.
    - `return_atom`: An `AtomNode` (for functions only).
- **`BodyNode(locals, algorithm)`**:
    - `locals`: A `Max3Node` (for local variables).
    - `algorithm`: An `AlgorithmNode`.
- **`Max3Node(variables)`**: Contains a `list` of 0 to 3 `VarNode` objects.
- **`MainProgNode(locals, algorithm)`**: Similar to `BodyNode` but for `main`.
- **`AlgorithmNode(instructions)`**: Contains a `list` of instruction nodes (e.g., `AssignmentNode`, `IfBranchNode`).
- **`AssignmentNode(variable, rhs)`**:
    - `variable`: A `VarNode`.
    - `rhs`: Can be an `AtomNode`, `FunctionCallNode`, or `ParenTermNode`.
- **`IfBranchNode(condition, then_branch, else_branch)`**:
    - `condition`: A `TermNode`.
    - `then_branch`: An `AlgorithmNode`.
    - `else_branch`: An `Optional[AlgorithmNode]` (it will be `None` if no `else` part exists).
- **`WhileLoopNode(condition, body)`** / **`DoUntilLoopNode(body, condition)`**: Self-explanatory.
- **`ParenTermNode(term)`**: Represents an expression in parentheses.
    - `term`: Will be either a `UnaryOperationNode` or `BinaryOperationNode`.
- **`BinaryOperationNode(left_operand, operator, right_operand)`**:
    - `operator`: A string (e.g., `'plus'`, `'eq'`).
    - `left_operand`, `right_operand`: `TermNode` objects.
- **`AtomNode(value)`**: A leaf for expressions.
    - `value`: Will be either an `int` (for numbers) or a `VarNode` (for identifiers).
- **`FunctionCallNode(name, arguments)`** / **`ProcedureCallNode(name, arguments)`**:
    - `name`: A `VarNode`.

- `arguments`: An `InputNode`.
- **`InputNode(arguments)`**: Contains a `list` of 0 to 3 `AtomNode` objects.

---

6. Key Grammar & Disambiguation Logic

The parser is built from the LL(1) grammar (`LL1.md`), which solves several ambiguities from the original spec:

1. **Dangling-Else:** Solved with `BRANCH → 'if' ... BRANCH'` and `BRANCH' → 'else' ... | ε`. The `_parse_branch()` function *always* calls `_parse_branch_prime()`, which then *looks* for an `else` token. If `else` is not found, `_parse_branch_prime()` returns `None`.
2. **Assignment vs. Procedure Call:** Solved with `INSTR_AFTER_ID`. When an instruction starts with an `id`, `_parse_instr()` calls `_parse_instr_after_id()`. This helper function looks ahead one token:
   - If it sees `(`, it parses a `ProcedureCallNode`.
   - If it sees `=`, it parses an `AssignmentNode`.
3. **Assignment RHS (Identifier vs. Function Call):** Solved with `ASSIGN_RHS` and `ASSIGN_RHS_ID'`. When the right side of an assignment starts with an `id`, the parser checks the *next* token:
   - If it sees `(`, it parses a `FunctionCallNode`.
   - If it sees anything else (like `;` or `}`), it chooses the epsilon rule and the `id` is treated as an `AtomNode`.

---

7. Next Steps for Semantic Analysis

1. **Implement the Symbol Table (Task 4):** Use a data structure (like a hash map of stacks) that can manage SPL's static, nested scopes (Global, Main, Proc-local, Func-local).
2. **Create a Tree Crawler (Task 5 & 6):** Write a new algorithm that recursively "walks" the AST.
3. **Populate Symbol Table:** As the crawler enters nodes like `ProgramNode`, `ProcedureDefNode`, `FunctionDefNode`, and `MainProgNode`, it should "push" a new scope onto the symbol table. As it encounters declarations (`VariableDeclsNode`, `Max3Node`), it should add those variables to the current scope.
4. **Perform Scope Checking (Task 5):** Use the symbol table to enforce the rules from `Phase 2 - SPL_Scopes.pdf` (e.g., no duplicates in the same scope, no shadowing parameters, all used variables are declared).

5. **Perform Type Checking (Task 6):** Use the symbol table to look up the types of variables and check them against the rules in `Phase 3 - SPL_Types.pdf` (e.g., `if` condition must be boolean, operands of `plus` must be numeric). You should annotate the AST nodes with their types as you go.

The `parser_spl.py`, `ast_nodes.py`, and `LL(1).pdf` files provide the complete structure you will be working with.