Task 1 - 3: *19th*
Task 4 - 6: *24th / 26th*
Task 7 - 9: *27th / 31st*

Parser

- **Task 1: Finalize AST Node Structures**
  - Create the data structures for every node type the AST will have (e.g., `IfNode`, `AssignNode`, `PlusNode`).
- **Task 2: Check LL1 Grammar Conversion**
- Ensure:
  - **Ambiguity Eliminated**:"dangling-else" ambiguity:
    - `BRANCH ::= if TERM { ALGO }`
    - `BRANCH ::= if TERM { ALGO } else { ALGO }`
  - **Left-Factorization Performed**: Factor out the common prefix `if TERM { ALGO }` to make the choice deterministic for an LL1 parser
  - **Left-Recursion Eliminated**: Check for any remaining left-recursion (though the SPL grammar seems to favor right-recursion, which is good for LL1).
- **Task 3: Implement the Recursive Descent Parser**
  - Compute First, Follow, Lookahead and closure algorithmically
  - Write one parsing function for each non terminal in the new LL1 grammar (e.g., `parse_algo()`, `parse_term()`).
  - Inside each function, use the next token (lookahead) to decide which production rule to follow.
  - As the parser executes, it should build and return the corresponding AST node.

Semantic Analysis: scopes and types

traverses AST to check for context sensitive errors and gather information for the code generator.

- **Scoping Rules**
  - **Static Scoping**. Spec says static, all name resolution can be done at compile time.
- **Task 4: Implement the Symbol Table**
  - **Data Structure**: Since SPL has nested scopes (global, main, proc/func locals), a hash map of stacks implementation seems plausible. Pushing a new map on scope entry and popping it on exit naturally handles static scoping.
  - **Information to Store**: For each identifier, the symbol table should store its **type** (`numeric`, `boolean`, `string`, or `type-less` for procedures) and its **scope level**. We'll also need to store a unique, generated name for use in the IR phase (mapping user-defined `x` to a unique `vx`).

- **Task 5: Implement the Scope Checker**
    - Create a tree crawling algorithm (recursive visits) that traverses the AST.
    - This crawler will use the symbol table to enforce the rules from `Phase 2: SPL_Scopes.pdf`:
        - No duplicate declarations within the same scope (e.g., in `VARIABLES` or `MAXTHREE`).
        - No shadowing of parameters by local variables .
        - No name clashes between variables, procedures, and functions in the global scope 4.
        - Verify that every variable used has been declared in a valid scope (local -> parameter -> global).
- **Task 6: Implement the Type Checker**
    - Create another tree-crawling pass (or combine it with the scope checker). This pass implements the rules from `Phase 3: SPL_Types.pdf`.
    - It uses the (now validated) information in the symbol table to check types. The output of this phase is an **annotated AST**, where each expression node is decorated with its type (`numeric` or `boolean`).
    - Key checks include:
        - The condition (`TERM`) of `if`, `while`, and `until` must be of type `boolean`.
        - The operands of `plus`, `minus`, `mult`, `div` must be `numeric`, and the result is `numeric`.
        - The operands of `and`, `or` must be `boolean`, and the result is `boolean`.
        - The variable on the left side of an assignment must be `numeric`.

Intermediate Code Generation

Translate validated and annotated AST into a ir representation

- **Decision: IR Format**
    - **Choice: Three-Address Code**. As per the project specs, use a format where each instruction has at most one operator (e.g., `t1 = v1 + v2`).
- **Task 7: Implement Translation Functions**
    - Write a recursive tree-crawling code generator (`translate(node)`) that emits IR instructions for each AST node.
    - **Translate Expressions (`TERM`, `ATOM`)**: Flatten expression trees into a sequence of instructions, using temporary variables (`newvar()`) for intermediate results.
    - **Translate Control Flow (`if`, `while`, `do-until`)**:
        - Implement the custom logic from `Phase 4: code-gen.pdf`. Remember:
            - Use `REM L_name` for labels.

        - The `if-then-else` structure must be translated without an `ELSE` keyword in the target code, using a GOTO to jump over the `else` block's code.
    - **Translate Logical Operators (`and`, `or`, `not`)**:
        - Implement the **"cascading" (short-circuiting)** logic using conditional jumps, as specified in the instructions13.
        - For `not(TERM)`, translate it as if it were `TERM`, but swap the true and false branches/labels.
    - **Translate Function/Procedure Calls**
        - For now, simply generate a placeholder `CALL` instruction. The spec states that these will be handled by **inlining** later.
        - Example for `x = myfunc(y)`: `t1 = CALL myfunc(v_y)`, then `v_x = t1`.

IR to BASIC

Convert simple IR into executable BASIC code.

- **Task 8: Implement Function Inlining**
    - Before emitting BASIC, process the IR to handle the `CALL` instructions.
    - For each `CALL` to a non-recursive function/procedure, replace it by:
        - Generating assignment instructions to pass parameters.
        - Copying the translated body of the called function/procedure into the call site.
        - For functions, replace the `return ATOM` instruction inside the copied body with an assignment to the target variable of the `CALL`.
    - In lining process from slides .
- **Task 9: Implement the BASIC Emitter**
    - Two passes over the inlined IR.
        - **First Pass (Label Mapping)**: Iterate through the IR without generating code. Create a map that associates every symbolic label (`L0`, `L1`, etc.) with a future line number (e.g., `L0 -> 140`, `L1 -> 180`).
        - **Second Pass (Code Emission)**:
            - Iterate through the IR again, this time generating the final BASIC code.
            - Prepend a line number to each instruction, incrementing by 10 (e.g., `10`, `20`, `30...`).
            - For `GOTO Lx` or `IF ... THEN Lx`, look up `Lx` in the label map and emit the corresponding line number.
            - All other instructions are likely a direct mapping.
            - Write the final numbered lines to a `.txt` output file.

```
 ___________________________.
|;;|                     |;;||
|[]|---------------------|[]||
|;;|                     |;;||
|;;|                     |;;||
|;;|                     |;;||
|;;|                     |;;||
|;;|                     |;;||
|;;|                     |;;||
|;;|_____________________|;;||
|;;;;;;;;;;;;;;;;;;;;;;;;;;;||
|;;;;;;_______________ ;;;;;||
|;;;;;|  ___         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |;;;|         |;;;;;||
|;;;;;| |___|         |;;;;;||
\_____|_______________|_____||
 ~~~~~^^^^^^^^^^^^^^^^^~~~~~~
```