Project Title: Scoping Rules and Type Checking in a Custom Language Parser

Project Description

In this project, you will implement a parser for a custom programming language. The language has a set of rules governing variable declarations, type compatibility, and scoping, similar to many modern programming languages. The goal is to enhance your understanding of fundamental programming language concepts such as scoping rules, type checking, and parser design. You will implement various functionalities to check for errors like type mismatches, scope violations, and variable redeclarations. Your implementation will emit error messages to provide feedback on the parsing process, ensuring adherence to the language's rules.

This project is an extension of Project 1, where you initially built a basic parser for a custom language. Now, you will extend that parser to include additional features related to scoping and type checking. You can use your Project 1 code to extend the functionality. If you did not complete Project 1, the instructor's solution will be provided as a starting point.

Learning Objectives

- Understand the concept of lexical analysis and syntactic analysis.
- Learn how to implement a symbol table for managing variable declarations and scoping.
- Gain a solid understanding of type checking in programming languages.
- Develop an understanding of error handling and diagnostic messages for syntax analysis.

Updated Language Grammar

Statements

- `statement ::= assign_stmt | if_stmt | while_stmt | expr_stmt | function_call | decl_stmt`

Declaration Statement

- `decl_stmt ::= type IDENTIFIER '=' expression`

Types

- `type ::= 'int' | 'float'`

If Statement

- `if_stmt ::= 'if' boolean_expression '{' block '}' ('else' '{' block '}')?`

While Statement

- `while_stmt ::= 'while' boolean_expression '{' block '}'`

Factors

- factor ::= NUMBER | FNUMBER | IDENTIFIER | '(' expression ')'

FNUMBER

- `FNUMBER ::= [0-9]+\.[0-9]+`

Rules to Implement

1. Type Compatibility

No expression should involve variables or constants of different types. For instance, adding an `int` to a `float` should raise a **type mismatch error**. Your parser must ensure that all operations have compatible operand types. Only check type mismatch for `int` and `float`. Please ignore all other types of mismatches, such as `(None, int)`, `(None, float)`, `(None, None)`, `(bool, int)`, `(bool, float)`, `(bool, None)`.

Example:

```
int a = 10.2  // Type Mismatch between int and float
float b = 5.2
```

2. Variable Declaration Before Use

A variable must be **declared** before it is used in an expression or statement. You should track variable declarations using a **symbol table** and check each variable usage to ensure it has already been declared.

Example:

```
int a = 10
a = b  // Variable b has not been declared in the current or any enclosing scopes
```

3. No Redeclaration in the Same Scope

Variables cannot be declared multiple times in the same scope. Attempting to declare a variable with the same name in the same scope should raise a **redeclaration error**. However, redeclaration is allowed in different (nested) scopes.

Example:

```
int a = 10
int b = 5
int a = 5  // Variable a has already been declared in the current scope
if a > b {
  float a = 3.5  // This is allowed as it is in a new (nested) scope
}
```

4. Scoping Rules

Implement **block scoping** for variables. A variable declared inside a block should not be accessible outside of that block, but it should be accessible within the block and its nested blocks.

Example:

```
if 1 > 2 {
  int a = 10
  {
    int b = 20
    a = a + b  // Valid: b is accessible within the nested block
  }
  b = 5  // Variable b has not been declared in the current or any enclosing scopes
}

a = 15  // Variable a has not been declared in the current or any enclosing scopes
```

Project Workflow

Step 1: Lexer Implementation

- Implement the lexer to tokenize the input string. Ensure that keywords, operators, braces, and identifiers are all correctly identified.
- Handle numeric literals, including distinguishing between `int` and `float` types.

Step 2: Parser Implementation

- Start by creating functions to parse each type of statement according to the grammar.
- Implement **scope management** using helper functions `enter_scope()` and `exit_scope()`.

Step 3: Error Handling and Type Checking

- Add **type checking** logic to ensure type compatibility in expressions and assignments.
- Use the **symbol table** to track variable declarations and detect errors related to scoping.

Step 4: Testing

- Implement a set of **test cases** to validate your parser. Test cases should cover type mismatches, variable redeclarations, scope violations, and correct code.
- We will provide 8 test cases for you. In addition, there will be 3 hidden test cases in Gradescope.

Expected Output
Your parser should generate an **AST** for correct input and emit descriptive error messages for invalid code. The error messages should be emitted in the order that the errors occur in the source code. Ensure that you test your implementation thoroughly to handle all types of inputs as defined by the rules. We only test your output messages.

Submission Instructions

- Submit your `Parser.py` file only, not the entire `.zip` file.
- Your submission will be evaluated on **Gradescope**, and there will be **three hidden test cases** to assess the robustness of your implementation.
- Make sure your code is well-documented, with comments explaining key parts of the implementation.

Additional Notes

- This project is meant to challenge your understanding of compiler design fundamentals. Don't hesitate to reach out if you have questions regarding the implementation.
- You are encouraged to write clean, modular code. Break down the parsing logic into smaller, reusable functions where possible.

Academic Integrity Course Policy

Dishonesty includes but is not limited to cheating, plagiarizing, facilitating acts of academic dishonesty by others, having unauthorized possession of examinations, submitting work of another person, or work previously used without informing the instructor.

Students who are found to be dishonest will receive academic sanctions and will be reported to the University's Office of Student Conduct for possible further disciplinary sanctions; refer to Procedure G-9:
http://undergrad.psu.edu/aappm/G-9-academic-integrity.html.

Furthermore, this course will follow the academic sanctions guidelines of the Department of Computer Science and Engineering, available at:
http://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx.

For violations of academic integrity, we will apply the following sanctions:

- **0** for the submission that violates academic integrity
- A **reduction of one letter grade** for the final course grade

Good luck, and happy coding!