Small Programming Language Project Instructions

1. Project Introduction

Project Title: Writing an Interpreter

Main Objective: Build a parser and interpreter for a simple programming language with arithmetic, variables, and functions.

General Skills:

- Practice with version control systems
- Group work collaboration
- Implementation of software design patterns
- Open-ended project planning

Specific Skills:

- Building understanding of how programming languages are implemented
- Learning about *lexers, symbol tables, and abstract syntax trees*.

Optional "Stretch" Skills:

- Learning about static type checking
- Advanced error/exception handling

2. General Requirements & Restrictions

Team Structure

- Teams will have 3 members
- All members must contribute, verified by peer evaluations and GitHub history

Version Control

- Teams must use GitHub project management tools
- Projects should be broken down into discrete tasks. These tasks will be managed & assigned using GitHub and tracked from planning through completion
- Regular commits/pull requests from feature branches expected

Technical Requirements

- All code must be written in Java (firm requirement)

Timeline

- First proposal due: March 7
- Feedback provided by: March 19
- Required check-in meeting by: April 11 (to review planning & design choices)
- Project code due: May 12
- Project presentations to TAs: Until May 13 (can be scheduled after May 1 with group & TA discretion)

Documentation

- All code must be documented with comments and tests
- Fully featured JavaDocs included as optional "completeness" feature
- Work with outside resources (books, websites, LLMs/Copilot) is permitted but must be cited

3. Specific Requirements for this Project

Technical Components

Students will be responsible for producing a program called `SPROLARunner.java` (**S**mall **Pro**gramming **La**nguage) which takes in a command-line argument. This argument will be the name of a file that contains an example of a SPROLA program which will be **lexed**, **parsed**, **interpreted**, and **executed.**

Lexer

This is a program that converts a file into a series of *lexical tokens*. Lexers break source code down into the individual pieces that make up a program—identifiers, literals, brackets, function calls, etc.—without having any control over the behavior of the program. A lexer can be thought of as a kind of stream or iterator which returns successive tokens as long as they are available and complains when some content of the file is not compatible with the grammar (syntax) of the program.

Parser

The parser is a program that consumes tokens from the lexer and builds an *abstract syntax tree* (AST). An AST is a data structure like an expression tree which represents the semantic structure of the program. The AST is built by creating nodes for each of the tokens consumed and arranging them together so that the interpreter (next step) can traverse the tree and execute the underlying instructions.

Interpreter

The interpreter is a program that maintains a *symbol table* as it traverses through an AST to execute the instructions that make up the program. The interpreter will actually perform the computations and instructions laid out in the original program.

Error Handling

All errors should be reported with clean, descriptive printed messages that include:

- Syntax Errors
  - Incomplete expressions
  - Invalid token sequences
- Variable Reference Errors
  - Using variables before assignment
  - Using variables outside of their scope
- Function Reference Errors
  - Calling undefined functions
  - Wrong number of arguments

The messages can include as much or as little information as you like, but they should describe at a minimum the kind of problem encountered. No exception handling is necessary.

User Interface

The final program should be runnable by executing `java SPROLARunner <my_file>`. The file should be executed end-to-end without further intervention.

Required Project Features

- Native Java implementation of Lexer, Parser, and Interpreter
- Use of two or more design patterns
- Execution as noted above.

Optional Features

- Exception handling
- Implementation of text types or array types
- Type checking (i.e. identifying errors about mismatched function inputs or invalid operations for added types at the *parser* level rather than at the *interpreter* level in order to prevent programs crashing while running.)

4. Evaluation Scheme

Code Quality

- 80% testing coverage required

- Commented code required

Documentation

- Design document with class diagrams for all included classes

Presentation

- Required sections:
    - Project design
    - Initial project plans
    - Choices made for optional requirements
    - Project demo
    - Challenges faced/project retrospective
    - TA Q&A

Team Assessment

- Peer evaluation via Google form

5. Other Resources

- Visitor Design Pattern, potentially useful for interpreting an AST
- Factory Design Pattern, potentially useful for creating Nodes in the AST
- State Design Pattern, potentially useful for building the interpreter
- Book about Compilers, freely accessible through Penn Libraries.
    - You're writing an *interpreter,* not a *compiler,* so not everything is relevant. Should be helpful for understanding lexers, parsers, ASTs, symbol tables, etc.