Section 1. Introduction

This software package is designed to parse and evaluate expressions in Moe, our Rust-based language. The core functionality revolves around processing expressions represented in S-expression format (sexp) and evaluating their meaning based on predefined operations such as arithmetic, conditional logic, function applications, and references. This software allows for expressions involving integers, floating-point numbers, identifiers, arithmetic operations, lambda functions, and conditionals, with proper parsing and formatting mechanisms.

The package is structured with a parser to transform input strings into abstract syntax tree (AST) representations and an evaluation engine that processes these expressions. The internal data structures support complex expression types, such as function applications and conditional branches. The software is intended for use in symbolic computation, and formal testing will follow internal validation to ensure the reliability of parsing and execution mechanisms.

Section 2. Items to Be Tested

- **Parsing S-Expressions:** Does the `parse()` function correctly convert valid S-expression strings into their respective AST representations? Does it handle malformed input appropriately, returning errors for invalid structures?
- **Arithmetic Operations:** Do the `Plus` and `Mult` variants correctly parse and evaluate their operands? Are nested operations handled appropriately?
- **Conditional Expressions:** Does the `If` expression correctly evaluate the condition and return the expected branch? Are errors raised when conditions are not boolean?
- **Function Application:** Does the `App` expression correctly apply lambda functions to arguments? Does it handle undefined or invalid function applications gracefully?
- **Reference and Dereference Operations:** Do the `Ref`, `MutRef`, `Unbox`, and `Deref` operations correctly handle references to boxed values? Are mutable references correctly enforced, and is dereferencing appropriately checked?
- **Equality and Boolean Operations:** Do the `Eq` and `Bool` expressions function as expected, evaluating boolean values and checking for numeric equality between operands?

Section 3. Items Not Tested

No items in this module have had professional testing except for those built into the Rust language.

Section 4. Tests to Be Conducted

1. **Parsing Valid Integer Test:**
   Test the `parse()` function with valid integer input (e.g., 1, 42) to ensure it correctly generates the appropriate `Exp::Int` objects. The program passes if all valid integer inputs are correctly parsed into `Exp::Int`.

2. **Parsing Invalid S-Expression Test:**
   Test the `parse()` function with malformed S-expressions (e.g., unmatched parentheses, invalid tokens) to verify that the parser returns appropriate `ParseError`. The program passes if all malformed inputs result in errors.

3. **Arithmetic Operations Test:**
   Test the parsing and evaluation of `Plus` and `Mult` expressions. This test will use simple and nested arithmetic expressions (e.g., `( + 1 2 )`, `( * 2 ( + 1 3 ))`) to ensure correct evaluation. The program passes if all valid expressions are evaluated accurately.

4. **Conditional Expression Test:**
   The `If` expression will be tested with both true and false conditions to check if the appropriate branch is selected. It will also test erroneous conditions that are not boolean. The program passes if all valid expressions evaluate correctly, and non-boolean conditions raise errors.

5. **Lambda Function Application Test:**
   Test lambda creation and application by parsing and applying lambda functions with single parameters (e.g., `(lambda x (+ x 1))`). The program passes if valid applications of lambdas produce the expected results and errors occur for invalid applications.

6. **Reference Operations Test:**
   Test the handling of reference expressions by creating and dereferencing values (e.g., `box`, `unbox`, `ref`). The program passes if the correct values are boxed, unboxed, and dereferenced, and errors are raised for invalid operations.

7. **Equality Operation Test:**
   The `Eq` expression will be tested to verify that numeric equality comparisons work correctly for both integers and floats. The program passes if valid equality checks succeed and mismatches are appropriately flagged.

Section 5. Environment Needed for the Test

Hardware:

The software will be tested on the same hardware that was used for development. This ensures compatibility and consistency with the environment in which the code was written.

Software:

The software will be tested using the same software setup from development. Other software and files needed include:

- **Rust compiler and toolchain (Cargo):**
  The Rust compiler and the Cargo build system will be required to compile and run the program, ensuring that the correct environment is used during development and testing.
- **Unit testing framework (Rust `test`):**
  The built-in Rust testing framework will be used to create unit tests for functions in `src/parse.rs` and other modules. The framework will verify that the correct outputs are generated for given inputs and that errors are handled appropriately.
- **Integration test suite:**
  A set of integration tests that will run using Cargo's built-in test runner (`cargo test`) to ensure that all parts of the program work together correctly.
- **Test driver for parsing and validation:**
  A driver will input various valid and invalid syntax/grammar for the parsing test to ensure the `parse.rs` logic behaves as expected. This includes valid statements, missing/invalid tokens, and edge cases for expressions.
- **Test driver for error handling:**
  This driver will pass incorrect input to ensure the correct error messages are displayed and error recovery mechanisms work as intended.
- **Documentation generation (`cargo doc`):**
  The Rust documentation tool will be used to verify that all code is properly documented and that documentation is generated without errors.

Section 6. Test Deliverables

A detailed report will be generated for each test indicating the success or failure of each test. Each report will include a description of the test, the test's output, and any relevant data files. In instances where the case fails, the report will include the reason for the test's failure.
The source code and/or algorithm for each test will be included with the documentation.

Section 7. Responsibility for the Tests

Kevin Skinner will be responsible for implementing and conducting all test cases, including input validation, parse unit testing, and data handling tests.