Borrow checking LISP language

Summary

This programming language is a project designed to bring Rust's memory safety principles and ownership model into a functional, LISP-style language. Built entirely in Rust, this language leverages move semantics and borrow checking to create a safe, flexible, and performant tool for functional programming without relying on garbage collection.

Problem Statement

Languages today often face a trade-off between functional flexibility and memory safety. Many functional languages rely on garbage collection, which adds a fairly significant performance overhead. Some languages like C/C++ are not garbage collected so they end up being much faster. But they trade memory safety for that performance. Rust bridged this gap by implementing memory safety without the need for a garbage collector. However, this novel method of memory management put forward by Rust is yet to be brought to the world of functional progamming. This new language seeks to close this gap by: 1. Ensuring memory safety without the need for a garbage collector. 2. Providing functional flexibility through a LISP-based syntax. 3. Enabling static analysis for early detection of memory and type errors, streamlining runtime execution. 4. Providing faster runtime speed by removing safety checks at runtime, and relying on assumptions proven by the static analyzer.

Language Design Philosophy

The design emphasizes **memory and type safety** with Rust-inspired static checks, **functional programming patterns** with a LISP-based syntax, and **high performance** by enabling the interpreter to assume static analysis guarantees. The language's design centers on the following principles: - **Ownership and Borrowing**: Explicit memory management allows safe, predictable memory use without runtime overhead. - **Immutability by Default**: Enforces strict rules for mutable references, ensuring safe parallel and sequential execution. - **Simple, Consistent Syntax**: S-expressions provide an intuitive structure that's easily parsed and extensible. - **Error Prevention**: Comprehensive static analysis, including type and borrow checking, catches potential errors before runtime, minimizing unexpected failures.

Key Features

- **Move Semantics**: Ownership and move semantics control variable scope and reassignment, ensuring no unintended data sharing in mutable contexts, as well as reducing costly memory copy operations.

- **Borrow Checking**: Enforces safe access patterns for mutable and immutable references, eliminating memory errors without requiring garbage collection.
- **Static Analysis**: Performs compile-time checks for type consistency, borrow validity, and safe memory access, allowing the interpreter to make efficient assumptions.
- **Functional Expression Evaluation**: Supports lambda functions, closures, and scoping for functional programming flexibility.
- **LISP-inspired Syntax**: Enables straightforward recursive descent parsing and supports LISP-style operations like conditional expressions, arithmetic, and lambda expressions.

Target Applications

This language is intended for applications where memory safety and performance are crucial, and a functional language is preferable. Examples include: - **Compiler and Interpreter Construction**: Its support for memory-safe references and functional design make it perfect for building performant compilers, interpreters, and language tooling. - **Educational Tools**: Aiming to teach Rust-inspired memory management concepts within a more simple, accessible syntax. - **Data Analysis and Lightweight Scripting**: Tasks requiring lightweight, memory-safe scripting capabilities without the burden of complex runtime management. - **Data Processing Pipelines**: Functional programming with memory safety is ideal for efficient and safe data transformation pipelines.

Technical Implementation

1. Parser

- Transforms LISP-style s-expressions into an abstract syntax tree (AST), structuring expressions into `Exp` nodes for interpretation.

2. Static Analyzer

- Enforces memory safety with move semantics and borrow checking by analyzing the AST.
- Includes type-checking for function applications, conditional logic, and arithmetic, ensuring robust error prevention.

3. Interpreter

- Recursively evaluates AST expressions, relying on static analysis guarantees to assume safe memory access, valid references, and type consistency.

4. Environment and Store

- Manages variables (symbols) and heap-allocated boxed values, supporting safe mutable and immutable references with enforced borrow rules.

This language offers a unique blend of functional expressiveness and memory safety, bridging the gap between flexible LISP syntax and the robust memory management of Rust.