Programming Language Specification Document

Language Overview

This language is a new Rust-based project combining functional programming with safe memory management inspired by Rust's ownership model. The language features LISP-inspired syntax and focuses on efficient, predictable memory handling without garbage collection, making it suitable for systems programming, embedded systems, and data processing. Core features include move semantics, borrow checking, and strong static analysis to prevent runtime memory errors.

Language Structure

This language actually only has a single syntactic structure, the expression. There are technically no statements, or code blocks. In fact, there is no technical distinction between keywords and operators. They all act like a normal function application. These structures do exist in the language, but only at a conceptual level. There is no special syntax for any of these.

Expression Structure

- **Basic expression syntax**: Statements are structured as s-expressions, enclosed in parentheses. Each expression is a list of more expressions. An expression can be another list of expressions, or an atom, which is a single symbol such as an identifier or value.

Reserved Words

Reserved Word	Purpose
`if`	Conditional branching
`lambda`	Anonymous function declaration
`begin`	Code block with sequential exec.
`let`	Variable binding
`let-rec`	Recursive variable binding
`box`	Heap allocation
`unbox`	Dereference box value
`&`	Immutable reference
`!`	Mutable reference
`display`	Print a value to the screen

Data Types

Type	Size	Description
`Int`	i64	Fixed-size integer
`Float`	f64	Double precision floating point number
`Bool`	Boolean	Represents `true` or `false`
`Closure`	Dynamic	Stores a function with an environment
`Box`	Dynamic	Heap-allocated, ownable value
`Ref`	Dynamic	Immutable reference to a boxed value
`MutRef`	Dynamic	Mutable reference to a boxed value
`Moved`	N/A	Represents a moved (invalid) value

Operators

Arithmetic Operators

Operator	Purpose	Example
`+`	Addition	`(+ 5 3)`
`*`	Multiplication	`(* 5 3)`
`-`	Subtraction	`(- 5 3)`
`/`	Division	`(/ 6 3)`

Comparison Operators

Operator	Purpose	Example
`=`	Equality	`(= 5 5)`
`>`	Greater than	`(> 5 3)`
`<`	Less than	`(< 3 5)`
`>=`	Greater than or equal	`(>= 5 5)`
`<=`	Less than or equal	`(<= 3 5)`

Control Structures

Selection Sequences Conditional expressions use the `if` function:

```
(if (= x 0)
    5
    6)
```

Repetition Sequences

The language does not have traditional loops, as it encourages recursion for repeated actions. Recursive functions can be defined with let-rec for this purpose.

Functions and Procedures

Anonymous functions (lambdas) and function application:

```
(lambda (x) (* x 2)) ; Defines a lambda that doubles its argument
((lambda (x) (* x 2)) 5) ; Applies the lambda to the argument 5
```

Tokens System

- **Token identification**: Tokens are identified by parsing s-expressions, with individual tokens representing literals, identifiers, operators, or reserved words. In a LISP language, tokens are also called atoms.
- **Token relationships**: Tokens are structured using nested lists (expressions within expressions), with relationships defined by the hierarchical tree structure of the AST.