Documentation

Getting Started

Dependancies

If you would like to build the program yourself, you will need rust and cargo installed. Install them by following the instructions on rustup

Installation

To install the language, download the executable from the project's repository or use the following commands in your terminal:

```
# Clone the repository
git clone https://github.com/GregShiner/cmpsc-470-final.git
cd cmpsc-470-final

# Build and run the interpreter in REPL mode
cargo run

# Build and run the interpreter with a file input
cargo run -- input.lisp

# Or compile the binary yourself first
# Install dependencies and compile the interpreter
cargo build --release

# Run the language interpreter in REPL mode
./target/release/cmpsc-470-final

# Run the language interpreter with a file input
./target/release/cmpsc-470-final input.lisp
```

Quick example

Just running the program without any arguments will put you into a REPL environment. However, you can very easily have it execute a file containing a program expression.

Create a file called `example.lisp` with the following contents:

```
(+ 1 2)
```

Then, if you have already compiled the binary:

```
./target/release/cmpsc-470-final example.lisp
```

Or, compile the interpreter and run it with:

```
cargo run -- example.lisp
```

This will print `Int(3)` to the console.

The interpreter will always output the result of the expression in debug format (`<type>(<contents>)`)

Reference

Basic syntax

Statements are written as s-expressions, enclosed in parentheses. Each expression can be an atomic value, an operation, or a nested expression. Code blocks are created with the begin keyword (NOTE: `begin` keyword interpreter has not been implemented yet).

Example:

```
(begin
  (display (+ 3 4))
  (display (+ 5 7)))
```

Output:

```
7
12
Int(12)
```

`begin` always returns the result of the last expression

Example:

```
(+ 5 (- 6 3))
```

Output:

```
Int(8)
```

Data Types Reference

- **Int**: Integer values.
- **Float**: Floating point values.
- **Bool**: Boolean values (true or false).
- **Closure**: Closure created from a `lambda` expression. Contains a body and captures its environment.
- **Box**: Heap-allocated values that support ownership and borrowing.
- **Ref**: Immutable reference to a Box.
- **MutRef**: Mutable reference to a Box.
- **Moved**: Represents a moved (invalid) value. (Only used for internal representation; cannot be constructed on its own)

Operators

Arithmetic Operators

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

NOTE: All arithmetic operators requires inputs to be either both ints or both floats. They will always return the same type as the input. Diving 2 integers will always do floor division.

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)`

NOTE: All comparison operators requires inputs to be either both ints or both floats. They will always output a `Bool` type

Control Structures

If Used for conditional branching

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

The first argument to `if`, the condition, must resolve to a `Bool`. The remaining arguments can be any type, as long as they are the same. `if` will return the value of the second argument if the condition is true, otherwise, it returns the third argument.

Begin Evaluates multiple expressions in a sequence, and returns the value of the last expression.

```
(begin
  (display "Step 1")
  (display "Step 2"))
```

Functions and Procedures

Functions are always anonymous lambda functions that can be applied by applying arguments

```
((lambda x (* x 2)) 5) ; Doubles the input, x. Returns Int(10)
```

Define recursive functions using let-rec (sugar not yet implemented, although still possible by manually using y-combinator)

```
(let-rec ((factorial (lambda (n)
                       (if (= n 1)
                           1
                           (* n (factorial (- n 1)))))))
  (factorial 5))
```

You can also create a value binding using the `let` syntactical sugar

```
(let (x 5) (* x 3)) ; outputs 15
```

Best Practices

- **Memory Management**: Boxed values should be used judiciously because while they are still more performant than garbage collected values, they are still heap allocated which is slower.
- **Mutibility**: Since mutable references cannot exist alongside other references to the same value. Creating mutable references leads to complex problems in scenarios where you need multiple references.
- **Begin**: Pure functions that do not rely on sequencial operations can be heavily optimized and paralellized by the interpreter. Using `begin` expressions reduces the opportunities for optimization heavily.

Grammar

```
<exp> ::= <int>
        | <float>
        | <id>
        | true
        | false
        | (+ <exp> <exp>)
        | (- <exp> <exp>)
        | (* <exp> <exp>)
        | (/ <exp> <exp>)
        | (lambda (<id>) <exp>)
        | (let (<id> <exp>) exp)
        | (<exp> <exp>)  ; function application
        | (if <exp> <exp> <exp>)
        | (= <exp> <exp>)
        | (> <exp> <exp>)
```

```
| (< <exp> <exp>)
| (>= <exp> <exp>)
| (<= <exp> <exp>)
| (begin <exp>*) ; (interpreter not yet implemented)
| (& <exp>)      ; immutable reference (interpreter not yet implemented)
| (! <exp>)      ; mutable reference (interpreter not yet implemented)
| (box <exp>)    ; (interpreter not yet implemented)
| (unbox <exp>)  ; (interpreter not yet implemented)
| (@ <exp>)      ; dereference (interpreter not yet implemented)
| (:= <exp> <exp>) ; set mutable reference (interpreter not yet implemented)
```