Sebastian Benjamin (benjamse)
Khuong Luu (luukh)
Phi Luu (luuph)

First Milestone Requirement (Due Thurs, Feb 27)
Final Requirement (Due Thurs, Mar 12)

Final Project: Lil' Lisp

I. Introduction

What is the name of your language?
Lil' Lisp

What is the language's paradigm?
Functional

(Optional) Is there anything especially unique/interesting about your language that you want to highlight upfront?
No

II. Design

What features does your language include? Be clear about how you satisfied the constraints of the feature menu. What level does each feature fall under (core, sugar, or library), and how did you determine this?
The levels (core, syntactic sugar, library) of the features below are annotated in the parentheses.

1. Mandatory Features

The following features are categorized as core features and are crucial to the expressiveness and safety of our language. We determined that all of the constructions currently listed in this document are Core because if any of these were to change or be removed, the list of programs that Lil' Lisp is able to express would change.

1.1. Basic Data Types and Operations (Core)

Basic data types:

- Boolean
- Integer
- String
- List

Boolean operations:

- Not (`not`)
- Equal to (=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

Integer operations:

- Addition (+)
- Subtraction (–)
- Multiplication (*)
- Division (/)

Using the basic data types and operations listed above, we can represent both the basic values themselves and the operations on these values.

1.2. Conditionals (Core)

Syntax:

```
(if (conditional-expression) (then-expression) (else-expression))
```

This construct provides branching in our language. All three expressions are required in this construct.

1.3. Loops/Recursions (Core)

Looping/iteration is provided via recursive function calls since our language is functional.

1.4. Names (Core)

Syntax:

```
(let ((var1name var1value) .. (varNname varNvalue)) (<s-expressions>))
```

This expression assigns value `var1value` to variable name `var1name` and so on. After that, the expression `(<s-expression>)` is evaluated.

`var1name`, `var2name`, ..., `varNname` are variable names and `var1value`, `var2value`, ... `varNvalue` are the initial values assigned to their corresponding variable names.

These satisfy the variables/local name constraint, as they are immutable local variables.

1.5. Functions with Arguments (Core)

Syntax:

```
(fn function-name (parameter-list) (body))
```

This construction allows us to abstract out repeated code and give it a name and a parameter list. These are passed into the expression evaluator at runtime and can be called with the `(call fn-name (arg1 … argv))` construction. The body of a function is allowed to contain a recursive function call.

Our scoping is **static scoping** with **closure implementation**.

2. Extra Features (At Least 3 Points)

2.1. Strings and String Concatenation Operation (1 Point) (Core)

Example of a String: `"I am once again asking for your financial support."`

To concatenate 2 strings:

```
(++ '<the first string value> '<the second string value>)
```

For example, the expression

```
(++ '"I am once again " '"asking for your financial support.")
```

is evaluated to the string value

```
"I am once again asking for your financial support."
```

This is a Core feature that helps Lil' Lisp create and concatenate string values. An example of a string value is "The helmet stayed on". None of the features from (1) to (5) can do this, so Lil' Lisp cannot handle string values if this feature is removed.

2.2. List/Array Data Type and Operations (2 Points) (Core)

List operations include indexing and concatenation.

Example of a list:
`(1 2 3)` is a list of 3 elements whose values are integers 1, 2, and 3.

To concatenate two lists:

`(++ <the first list value> <the second list value>)`

For example, the expression

```
(++ (1 2 3) (4 5 6))
```

is evaluated to the list value

```
(1 2 3 4 5 6)
```

To index an element in a list:

`(nth <index> <the list value>)`

For example, the expression

```
(nth 2 (1 2 3))
```

is evaluated to an integer value

```
3
```

This is also a Core feature, as these list operations are not defined by any of the aforementioned features.

What are the safety properties of your language? If you implemented a static type system, describe it here. Otherwise, describe what kinds of errors can occur in your language and how you handle them.

- This language does not implement a static type system. Instead, each expression checks its operands' types at runtime and returns an error string if any operand has an incorrect type.

- For instance, in the concatenation function, the first argument (either a list or a string) is used to require all arguments following to have the list type or the string type, respectively. If any following argument is not a list/string, the expression evaluator returns a "`Both operands must be a collection type`" error.
- This interpreter gives informative error feedback to the user. For example, an input of **`(I 4)`** to a **name** parameter that requires a type of **`(N name)`** doesn't crash the interpreter. Rather, it will give an error telling the user that the type is wrong: "`Name must be a string`"

III. Implementation

What semantic domains did you choose for your language?
Semantic domain: literal *`value`*
Where
`value` is either a boolean, an integer, a string, or a list.

How did you decide on these?
Our language, Lil' Lisp, is a (little) subset of the Lisp programming language. Lil' Lisp aims to be a general-purpose language like Lisp (without the IO). It transforms a set of operations (or values) into a resulting value. Since it's functional, it does not define sets of procedural operations that map to values, like in imperative languages--the program simply ends up producing a single value from a set of basic types: `bool`, `int`, `string`, and `list`. This set is, therefore, our semantic domain.

Are there any unique/interesting aspects of your implementation you'd like to describe?
We have in-built error handling in the `Expr` type so that every expression can return an error as an expression, rather than returning a `(Maybe Expr Error)` and having to error handle at each level of the expression evaluator.