Sebastian Benjamin (benjamse)

Khuong Luu (luukh)

Phi Luu (luuph)

First Milestone Requirement (Due Thurs, Feb 27)

Final Requirement

1. 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?

For now, no

2. 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?

Feature (Core, Syntactic Sugar, Library) is annotated in parentheses.

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. Basic data types and operations. (Core)

Basic data types: Boolean, Integer
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 (/).

Therefore, we can represent both the basic values themselves and operations on these values.

2. Conditionals. (Core)
(if (conditional-expression) (then-expression) (else-expression))

This construct provides branching in our language.

3. Loops/Recursion (Core)
Looping/iteration is provided via recursive function calls since our language is functional.

4. Names (Core)
(let ((var1name val1value) (var2name val2value).. (varname varvalue)) (<s-expressions>))

This expression assigns value *val1value* to variable name *var1name* and so on. After that, the expression (<s-expression>) is evaluated

var1name, var2name, ..varname are variable names and val1value, val2value, .. varvalue are the initial values assigned to their corresponding variable names.

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

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.

Extra features (at least 3 points)

6. Strings and string concatenation operation (1 point) (Core)
7. List/array data type. Operations include indexing and concatenation (2 points) (Core)

Extra features descriptions

6. **Strings and string concatenation operation** (1 point) (Core)

Example of a string: “I am once again asking you for financial support.”

To concatenate 2 strings:
(++ ‘<the first string value> ‘<the second string value>)

For example, the expression

(++ ‘"I am once again " ‘"asking you for financial support.")

will be evaluated to a string value “I am once again asking you for 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.

7. **List/array data type. Operations include indexing and concatenation** (2 points) (Core)

Example of a list:

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

Concatenate 2 lists: (++ <the first list value> <the second list value>)

For example, the expression

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

Will be evaluated to a list value (1 2 3 4 5 6)

Index in list: (nth <index> <the list value>)

For example, the expression

(nth 2 (1 2 3))
Will be 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.

- There is no static type system. However, each expression will check its operands types at runtime, and return a string error if any of operands are of the incorrect type.

- For instance, use either list or string as a first argument to concatenate function to indicate the following arguments after that are of type list or string, respectively. If either are not a list/string, the expression evaluator will return a “Both operands must be a collection type” error.

3. 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.