COMP 325 — Interpreter 4

Stateful Lists

Fall 2017

Abstract

For your fourth interpreter you get to skip the parser and work solely on the desugar and interp functions. We'll also set aside the user-interface to the language and not worry about how language users interact with these core functions. This includes ignoring the presence of top-level definitions. Your only concern is the desugaring and interpretation of expressions.

The main feature you're exploring with this assignment is the extension of the mutable box idea to a mutable pair. This extension allows us to capture the basis of mutable Lisp-like lists and linked-list like structures in general. Welcome back data-structures!

Expressions

The language should support the following expressions in the extended language.

- Binary Arithmetic Operators: `+`,`*`,`-`,`/`,`modulo`
- Numeric Comparison: `==`, `<`, `>`
- Boolean Operators: and,or,not (binary `and` and `or`. unary `not`)
- Basic `if` expression
- First Class, n-ary Functions: lambda expressions
- A `local` expression that is equivalent to letrec with multiple identifier bindings, i.e. it allows multiple (possibly) recursively local identifiers. Note that each recursive identifier can refer not only to itself, but also to one or more of the *other* recursive identifiers being defined.
- A null-type value `NIL`
- A mutable cons cell with the following:
  - `car` to select the first, and `cdr` to select the second
  - `set-car` to modify the first, and `set-cdr` to modify the second
- `is-cons` predicate to determine if a value is a cons cell or not
- `is-null` predicate to determine if a value is null or not

Semantics

Not much new is happening with the numeric and boolean operators other than adjusting for store-based interpretation. Use desugaring where possible and be certain that boolean operators short circuit. Functions in this language are strict in the number of arguments that they can take. No currying, no partial evaluation, n-ary functions in the core. The `local` expression is exactly the same as `letrec` and should similarly be desugared down to a single identifier binding form in the core. The `cons` pair is a natural extension to two elements of the single element box we studied in class. More details about `cons` are given below.

The interpreter should continue to use the environment to manage identifier scoping issues but now manage value storage and mutation through the store as discussed in chapter 31 and in class. Logically independent tests should not have any kind of dependency due to the store/state implementation.

Cons and Lists

In our language, a `cons cell` is a mutable pair. By including a `null-typed value`, `NIL`, along with this structure, we provide programmers the tools necessary for linked-list like data. The names we're choosing here are, as you know from the Lisp History paper, historical. The high-level semantics of each operation is discussed here, just in case you need a bit of a nudge or refresher:

- A `null` type value can be used as a stand in for an empty list. In this language, that is it's only use we'll consider. Historically, the keyword `NIL` is used to express a null type value. The `is-null` predicate recognizes a null type.

  ```
  (is-null NIL) is true
  ```

- The `cons` constructor function takes two arguments and builds a pair from them. So `(cons 1 2)` is the pair containing 1 and 2. The predicate `is-cons` recognizes a cons cell type value.

  ```
  (is-cons (cons 1 2)) is true
  (is-cons 1) is false
  (is-cons false) is false
  (is-cons NIL) is false
  ```

- The selectors `car` and `cdr` are the equivalent of `unbox`.

  ```
  (car (cons 1 2)) is 1
  (cdr (cons 1 2)) is 2
  ```

- The mutators `set-car` and `set-cdr` are the equivalent of `setbox`

  ```
  (local ( (c (cons 1 2)) )
    (begin
      (set-car c 4)
      (car c)))
  is
  4

  (local ( (c (cons 1 2)) )
    (begin
      (set-cdr c 4)
      (cdr c)))
  is
  4
  ```

Big-Picture Test

To do some big picture tests write up the following expression/program in its own check block and test that it produces the correct value.

```
(local ( (sum (lambda (lst)
                (if (is-null lst) 0
                    (+ (car lst) (sum (cdr lst)))) ))
    (sum (cons 1 (cons 2 (cons 3 (cons 4 NIL))))) )
```

Consider writing up some similarly classic list-based programs (an instance of a map, other folds, a mutation-based for-each, etc.) as tests as well or use them to tease out unit tests for the interp and desugar functions and their helpers. Just because you're not building a complete system doesn't mean the end goal can't and shouldn't inspire you and inform your work.

Logistics

For this assignment you only need to write the expression desugarer, interpreter, and any necessary helpers for those functions. No parser, no top-level definition handlers, no user-interface (i.e. run) function. The completed assignment is due on **11/08**. The grade is determined as follows:

Area	Points
Interp	25
Desugar	20
Data Definitions	10
Big Picture Test	5
Style and Comments	5
	65 total

You are expected to have all required expressions represented in all parts of your design. If it's not there and at least stubbed, then you lose points. At this point we should be able to lay out the skeleton of top level cases. Sufficiency of testing is covered for the section your testing. The core language should be as minimal as possible without going overboard. The quality of your sugar versus non-sugar choices will be evaluated as part of your Data Definitions grade. The style and comment part of your grade accounts for good coding practices like proper indentation, avoiding printed line wrapping, good identifier and function names, documentation, and commenting of logic.