Assignment: A store-passing interpreter for EXPLICIT-REFS  
CSC 312, “Programming language implementation”  
Department of Computer Science · Grinnell College  
September 21, 2015

This assignment is an extension of Exercise 4.12 from the Friedman and Wand book (page 112).

Part A: Stores without mutation

One weakness of the textbook’s implementation of the EXPLICIT-REFS language is that it uses a global variable, the-store, and destructively changes the value of that global variable every time a new reference is added and every time an assignment updates an old reference. This creates a dependency among all the procedures that use this global variable, making it necessary to keep track of the order in which those procedures are called and the “current” value of the-store at the time of each call in order to establish the correctness of the code.

This choice of representation for stores also suggests, misleadingly, that we need an implementation language that supports state and mutation in order to write an interpreter for a language that supports state and mutation. This is clearly incorrect, however. The equational specifications for EXPLICIT-REFS in section 4.2 exactly describe mutable references even though the mathematical notation itself doesn’t support any such thing. It does this by writing stores explicitly into the functions it defines, both as arguments and as values. We can do this in the implementation as well. This approach is called “store-passing.”

The value-of procedure at the heart of the interpreter will be laid out as in Figure 4.6 on page 116 of the textbook: Besides the expression to be evaluated and the environment that records the bindings, there will be a third argument, store, containing the values stored in the references that have already been created. value-of will return an Answer, which is a datatype (also defined in Figure 4.6) that contains both an expressed value and a store. The store contained in the answer that value-of returns may be different from the store it received as an argument; but we don’t have to destructively modify the older store to get the newer one. Instead, we can use our standard Scheme procedures, cons and car and cdr and the like, to create the new store without overwriting anything in the old one.

Incidentally, the signature for value-of in Figure 4.6 should read

\[ \text{value-of} : \text{Exp} \times \text{Env} \times \text{Sto} \rightarrow \text{Answer} \]

but some printings of the textbook show the result type, incorrectly, as ExpVal.

ExpVal is, however, the correct result type for the value-of-program procedure, which should extract the expressed value from the answer returned by the call to value-of and discard the store, which is no longer needed once the value of the program body has been computed.

The assignment is to complete the implementation of a store-passing interpreter for EXPLICIT-REFS using no Scheme expressions that have side effects on storage locations — no set!-expressions and no calls to vector-set! or set-car!, for instance.

Part B: Values of assignment expressions

In my implementation of EXPLICIT-REFS, every set-ref-expression has the same value, no matter which reference is changing or what is stored in that reference. In your implementation, make the value of every set-ref-expression the same value that the
expression is storing into the reference (like an assignment expression in C, which has the value of its right-hand-side subexpression).

**Part C: Side effects in subexpressions**

When the EXPLICIT-REFS interpreter evaluates an expression, it is possible for one of its subexpressions to change the store for another subexpression in ways that may surprise programmers. For instance, the value of the expression

```plaintext
let x = newref(143)
in -(begin setref(x, -(deref(x), 1)); deref(x) end,
      begin setref(x, -(deref(x), 1)); deref(x) end)
```

is 1 if the minuend of the diff-expression that is the body of the `let`-expression is evaluated before the subtrahend, but \(-1\) if the subtrahend is evaluated first. Note that my implementation of the EXPLICIT-REFS interpreter forces the minuend to be evaluated first.

That implementation of EXPLICIT-REFS supports only unary `proc`-expressions and single-operand procedure calls. Lift this restriction to allow `proc`-expressions with any number of parameters (including none) and calls with any number of operands. Since the evaluation of an operand expression might affect the store in which other operands will be evaluated, state explicitly the order in which the operands in a procedure call will be evaluated and ensure that your implementation actually achieves that ordering.

This assignment will be due on Monday, September 28. Please submit your solution in hard copy at the beginning of that day’s class session.