Homework 4

The file hw4.hs on the course web page contains an expanded version of the interpreter we have been looking at in class. Besides more syntactic constructs, this version contains state, output, nondeterminism, and errors all at once; this is reflected in the parameterized return type for the interpreter:

\[
\text{newtype All } a = \text{All } (\text{MyState} \to [((\text{String}, \text{Err } a), \text{MyState})])
\]

The given code has several holes which you must fill in for this assignment.

1. [30 points]
   Fill in the definitions of unit, bad and bind.

2. [20 points]
   Fill in the definitions of tick, getStep, out, and both.
   Note: the result of evaluating Out should be Unit.

3. [15 points]
   Fill in the definitions of getRef, setRef, newRef.
   Note: the result of evaluating e1 := e2 should be Unit; the result of evaluating MkRef e should be Ref i for some i.
4. [10 points]
We can add local definitions, Let ["v0" :=: e0, ...] e, where Var "v0" is bound to e0 in e (and so on). Let can be defined as syntactic sugar as follows:

\[
\text{let } v_0 = e_0, \ldots, v_n = e_n \text{ in } e \equiv (\lambda v_0 \ldots \lambda v_n.e) e_0 \ldots e_n
\]

Use the preceding transformation to fill in the interp clause for Let.

5. [30 points]
We can add recursion to this language by introducing a recursive local definition, LetRec. This definition can be seen as syntactic sugar for a recursive Let style unfolding as follows:

\[
(\lambda v_0 \ldots \lambda v_n.e)(\lambda u_0.e_0^*) \ldots (\lambda u_n.e_n^*) \leftrightarrow v
\]

\[
\text{letrec } v_0 = \lambda u_0.e_0, \ldots, v_n = \lambda u_n.e_n \text{ in } e \leftrightarrow v
\]

\text{ev\_letrec}

where \( e_i^* = \text{letrec } v_0 = \lambda u_0.e_0, \ldots, v_n = \lambda u_n.e_n \text{ in } e_i \).

Implement the interp clause for LetRec using the preceding evaluation rule. Note the form of the declarations is restricted to having lambdas on the right-hand side.

6. [20 points]
While \( e_1 \ e_2 \) represents a while–do loop where \( e_1 \) is the test condition
and e2 is the body of the loop. Implement the interp clause for While; note that the result of a while loop should be the value Unit. Hint: a While loop can be thought of as syntactic sugar for a LetRec construction.

The last section of hw4.hs contains a number of examples which you should use to test out your code; i.e. your code should correctly evaluate all of the given terms. In particular, the term fibImp, an imperative version of the Fibonacci function, uses both mutable references and While loops.

7. [30 points]
This last question lets you think a little about programming with functions and state.

Mutable references can be used to "memoize" a function, by associating a list of argument/result pairs with the function, as illustrated by the memo term in the examples section of hw4.hs. If f is a function (i.e. a lambda), memo :: f results in a term which behaves like f, but remembers previous applications, so that

Let ["mf" :=: memo :: f]

(mf :: Con 1 :& mf :: Con 2 :& mf :: Con 1)
results in f (Con 1) being evaluated once. However, memo does not recursively memoize functions, thus memo :: fib (where fib
Implement the `memorec` term which can be used to define recursively memoized functions. The definition of `fibm` illustrates how `memorec` is intended to be used.

The terms `fib` and `fibm` are provided to allow you to easily observe the effects of your memoization efforts.

is also from the examples section) still has an exponential running time.