☐ 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:

```
  newtype All a = All (MyState -> [((String, Err a), 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:

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

   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:

   $$\frac{(\lambda v_0 . \ldots \lambda v_n . e)\, (\lambda u_0 . e_0^*)\ \ldots\ (\lambda u_n . e_n^*) \ \hookrightarrow\ v}{\text{letrec } v_0 = \lambda u_0 . e_0, \ldots, v_n = \lambda u_n . e_n \text{ in } e \ \hookrightarrow\ v}\ \text{ev\_letrec}$$

   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 e1 e2` represents a while–do loop where `e1` 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

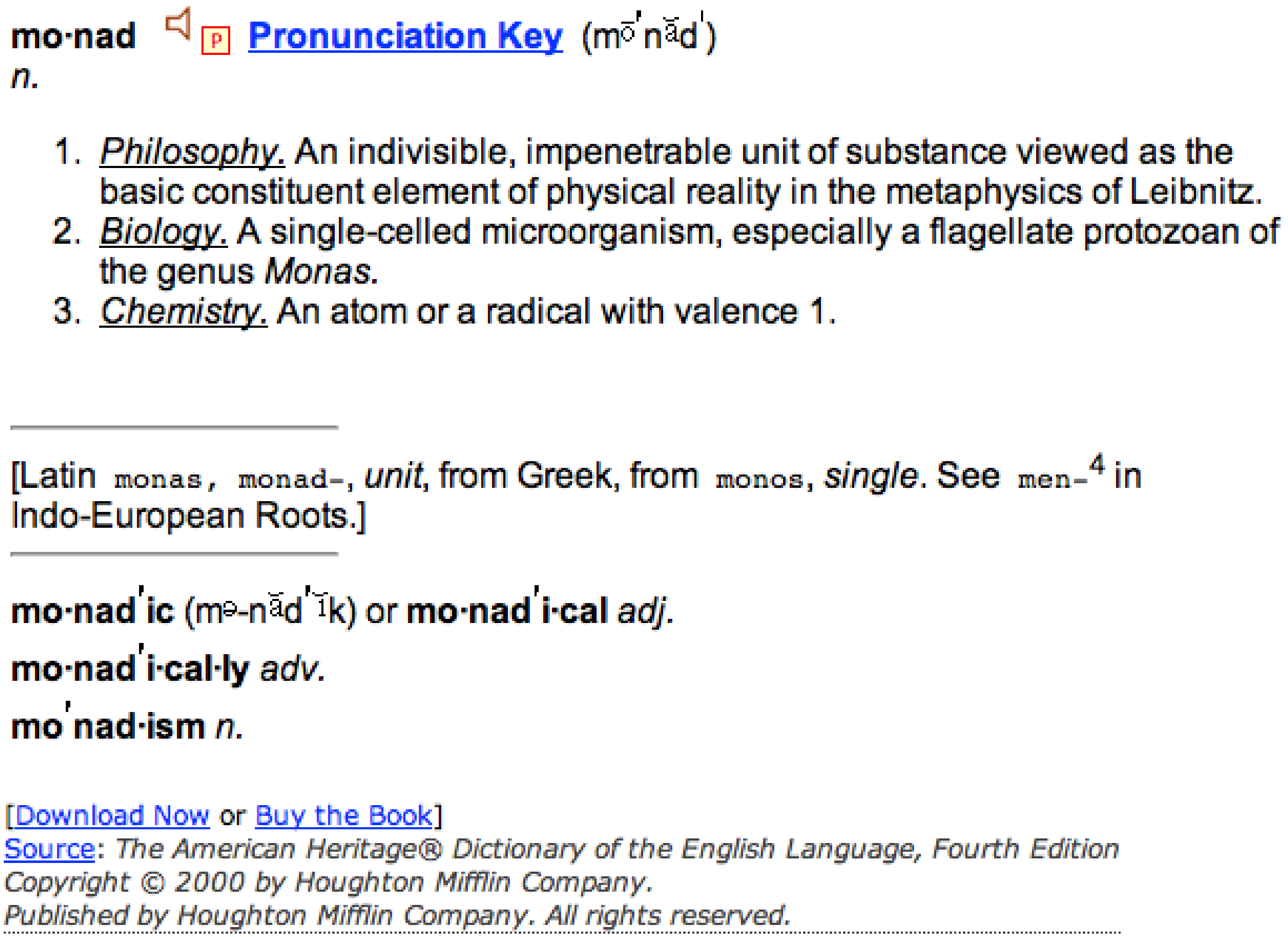
```
    Let ["mf" :=: memo :$: f]
         (mf :$: Con 1 :& mf :$: Con 2 :& mf :$: Con 1)
```

   results in `f (Con 1)` being evaluated once. However, `memo` does not recursively memoize functions, thus `memo :$: fib` (where `fib` is also from the examples section) still has an exponential running time.

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.

☐ What the hell are monads?



mo·nad Pronunciation Key (mō′năd′)
n.

1. *Philosophy.* An indivisible, impenetrable unit of substance viewed as the basic constituent element of physical reality in the metaphysics of Leibniz.
2. *Biology.* A single-celled microorganism, especially a flagellate protozoan of the genus *Monas*.
3. *Chemistry.* An atom or a radical with valence 1.

[Latin `monas`, `monad-`, *unit*, from Greek, from `monos`, *single*. See `men-`[4] in Indo-European Roots.]

---

mo·nad′ic (mə-năd′ĭk) or **mo·nad′i·cal** *adj.*

mo·nad′i·cal·ly *adv.*

mo′nad·ism *n.*

[Download Now or Buy the Book]
Source: *The American Heritage® Dictionary of the English Language, Fourth Edition*

□ History

- Monads are constructions from category theory.
- Category theory is ”abstract abstract algebra”.
- Category theorists like to
  - draw diagrams
  - wave their hands
  - elide lots of annoying, little details
- Category theory useful for theoretical computer scientists.
- Sometimes, category theory has more direct CS applications.

□ Monads

A monad is a type function M and a triple, $\langle \mathsf{map}, \mathsf{unit}, \mathsf{join} \rangle$, where (for all types A, B):

$$\mathsf{map} : (A \to B) \to MA \to MB \qquad \mathsf{unit}_A : A \to MA \qquad \mathsf{join}_A : MMA \to MA$$

and the following diagrams commute:

$$\begin{tikzcd}
MMMA \arrow[r, "\mathsf{join}_{MA}"] \arrow[d, "\mathsf{map}(\mathsf{join}_A)"'] & MMA \arrow[d, "\mathsf{join}_A"] \\
MMA \arrow[r, "\mathsf{join}_A"'] & MA
\end{tikzcd}
\qquad
\begin{tikzcd}
MA \arrow[r, "\mathsf{unit}_{MA}"] \arrow[dr, "\mathsf{id}_{MA}"'] & MMA \arrow[d, "\mathsf{join}_A"] & MA \arrow[l, "\mathsf{map}(\mathsf{unit}_A)"'] \arrow[dl, "\mathsf{id}_{MA}"] \\
& MA &
\end{tikzcd}$$

in other words:

$$\forall x : MMMA.\ \mathsf{join}_A(\mathsf{join}_{MA}(x)) = \mathsf{join}_A(\mathsf{map}(\mathsf{join}_A)(x))$$

$$\forall x : MA.\ \mathsf{join}_A(\mathsf{unit}_{MA}(x)) = x = \mathsf{join}_A(\mathsf{map}(\mathsf{unit}_A)(x))$$

or more succinctly:

$$\mathsf{join}_A \circ \mathsf{join}_{MA} = \mathsf{join}_A \circ \mathsf{map}(\mathsf{join}_A)$$

$$\mathsf{join}_A \circ \mathsf{unit}_{MA} = \mathsf{id}_{MA} = \mathsf{join}_A \circ \mathsf{map}(\mathsf{unit}_A)$$

□ Concrete Examples– Maybe

```
data Maybe a = Just a | Nothing

unit :: a -> Maybe a
unit x = Just x

join :: Maybe (Maybe a) -> Maybe a
join (Just x) = x
join Nothing = Nothing

map :: (a -> b) -> Maybe a -> Maybe b
map f (Just x) = Just $ f x
map f Nothing = Nothing
```

□ Concete Examples– List

```
unit :: a -> [a]
unit x = [x]

join :: [[a]] -> [a]
join [[]] = []
join (x:xs) = x++(join xs)

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

□ Kleisli Triple

A construction equivalent to a monad.

A Kleisli triple, $\langle M, \mathsf{unit}, \mathsf{bind} \rangle$, consists of a type function, M, and two associated functions (for any types A, B)

$$\mathsf{unit} : A \to MA \qquad \mathsf{bind} : (A \to MB) \to MA \to MB$$

and the following diagrams commute:

$$\begin{tikzcd} & MA \\ MA \arrow[ur, bend left, "\mathsf{bind}\,\mathsf{unit}_A"] \arrow[ur, bend right, "\mathsf{id}_{MA}"'] & \end{tikzcd} \qquad \begin{tikzcd} & MB \\ A \arrow[ur, "f"] \arrow[r, "\mathsf{unit}_A"'] & MA \arrow[u, "\mathsf{bind}\,f"'] \end{tikzcd} \qquad \begin{tikzcd} & MC \\ MA \arrow[ur, "\mathsf{bind}\,((\mathsf{bind}\,g)\circ f)"] \arrow[r, "\mathsf{bind}\,f"'] & MB \arrow[u, "\mathsf{bind}\,g"'] \end{tikzcd}$$

in other words:

$$\mathsf{bind}\,\mathsf{unit}_A = \mathsf{id}_{MA}$$

$$(\mathsf{bind}\,f) \circ \mathsf{unit}_A = f$$

$$\mathsf{bind}\,((\mathsf{bind}\,g) \circ f) = (\mathsf{bind}\,g) \circ (\mathsf{bind}\,f)$$

□ Kleisli Examples

```
-- Maybe
unit :: a -> Maybe a
unit x = Just x

bind :: (a -> Maybe b) -> Maybe a -> Maybe b
bind f (Just x) = f x
bind f Nothing = Nothing


-- List
unit :: a -> [a]
unit x = [x]

bind :: (a -> [b]) -> [a] -> [b]
bind f [] = []
bind f (x:xs) = f x ++ bind f xs
```

□ Equivalence of presentations

Monad, M and $\langle\text{map}, \text{unit}, \text{join}\rangle$, implies Kleisli triple, $\langle M, \text{unit}, \text{bind}\rangle$:

```
unit :: a -> M a        bind :: (a -> M b) -> M a -> M b
unit = unit             bind f x = join ((map f) x)
```

Kleisli triple, $\langle M, \text{unit}, \text{bind}\rangle$, implies Monad, M and $\langle\text{map}, \text{unit}, \text{join}\rangle$:

```
unit :: a -> M a        join :: M (M a) -> M a
unit = unit             join x = bind id x

map :: (a -> b) -> M a -> M b
map f x = bind (\y -> unit (f y)) x
```

□ Values and Computations

- Useful to distinguish values and computations,
  e.g. 2 is a value,
  (x = ref 1; x := !x + 1; !x)) is a computation.

- Computations might do many things,
  e.g. diverge, update state, produce output, etc.

- Monads offer a formal framework for distinguishing values and computations.
  - Each monad represents a different kind of computation.
  - A term of type `M a` is a computation of type `a`.
  - Types tell us which parts of program can cause effects.

- Monads offer an elegant guide for combining pure and impure language features.

□ Haskell Monad Class

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

It is convenient to switch the order of arguments for bind.

- Notice no constraints on behavior of `return` and `>>=`.
- Possible to define ”bad” `Monad` instances.
- Up to programmer to verify instances of `return` and `>>=` obey monad laws.
- Laws allow easier reasoning about monadic programs.

□ Example Haskell Monads

```
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing >>= f = Nothing

instance Monad [] where
    return x = [x]
    [] >>= f = []
    (x:xs) >>= f = f x ++ (xs >>= f)

newtype State s a = State (s -> (a, s))
instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State x) >>= f = State $ \s1 ->
        let (v1, s2) = x s1
            State v2 = f v1
        in v2 s2
```

☐ Monadically lifted functions

We can take non-monadic functions and lift them into any monad.

```
addM :: (Num a, Monad m) => m a -> m a -> m a
addM a b = a >>= \m ->
           b >>= \n ->
           return $ m + n
```

Haskell even supplies us with functions:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
   :

ap :: Monad m => m (a -> b) -> m a -> m b
```

□ do notation

Monads are so ubiquitous in Haskell that there is a special syntax:

```
do e         is equivalent to   e

do x <- e    is equivalent to   e >>= \x -> do c
   c

do e         is equivalent to   e >>= \_ -> do c
   c
```

```
addM a b = do m <- a
              n <- b
              return $ m + n
```