A Schemer's View of Monads
Partial Draft

Adam C. Foltzer & Daniel P. Friedman

December 2, 2011

Lecture 1: The State Monad

This tutorial lecture is based on the first four pages of "Notions of Computation and Monads" by Eugenio Moggi, who took the idea of monads from category theory and pointed out its relevance to programming languages.[1]

Everything in these two lectures will simply be purely functional code. There will be no **set!**s; there will be one *call/cc* to help motivate an example in the second lecture; and there will be lots of λs and **let**s. The only requirement is understanding functions as values.

The goal of these two lectures is to teach how monads work. It impedes understanding if we concern ourselves with a lot of details or sophisticated built-in tools, so we use only a very small subset of Scheme to expose the relevant ideas. There is one program written in continuation-passing style that shows one way of computing two values in one pass, but it is not important to understand the program. In fact, it is only necessary to notice a single occurrence of the symbol +. There is also the continuation monad, explained toward the end of the second lecture in section 8, and here, it might help to have some familiarity with first-class continuations.

1 Prologue: The Identity Monad

To start, we will walk through two typical Scheme programming challenges, and then show how they naturally give rise to a monad.

1.1 Recreating begin

Our first challenge is to recreate the behavior of Scheme's **begin** using only λ and function application.

```
> (begin (printf "One\n") (printf "Two\n"))
One
Two
```

In this example, **begin** enforces the order in which the two *printf* expressions are evaluated. To get the same behavior just from λ and application, we must take advantage of the fact that Scheme is a call-by-value language. That is, the arguments to a function are always evaluated before the body of the function. We need to arrange our expression so that (*printf* "One\n") is an argument to a function that contains (*printf* "Two\n").

[1] See `http://www.disi.unige.it/person/MoggiE/publications.html`.

```
> ((λ (_) (printf "Two\n")) (printf "One\n"))
One
Two
```

Success! Note that there's nothing special about _; it simply means that we do not care about the value of (*printf* "One\n"). We only care that it gets evaluated for its printing effect.

This code would look nicer if the evaluation order of our statements read from left-to-right, as with *begin*. The only reason our example reads the other way is the order of function application: (*function argument*). To get the order we want, we can define a backwards function application:

```
(define mybegin
  (λ (x f)
    (f x)))
```

```
> (mybegin (printf "One\n") (λ (_) (printf "Two\n")))
One
Two
```

This is an improvement. How about printing a third string?

```
> (mybegin (printf "One\n")
    (λ (_) (mybegin (printf "Two\n")
             (λ (_) (printf "Three\n")))))
One
Two
Three
```

1.2 Recreating let

Our next challenge is to recreate the behavior of Scheme's **let** using the same toolkit of λ and function application. Again, we start with a simple example.

```
> (let ([x 5])
    (+ x 3))
8
```

Nesting **let** in this way enforces the order of evaluation similarly to *begin*. Here, 5 must be evaluated before (+ *x* 3). We can start by applying the same basic structure as our **begin** example.

```
((λ (_) (+ x 3)) 5)
```

This isn't quite right, though, since **let** does more than **begin**. Rather than throwing away the value of, for example, (*printf* "One\n") by binding it to an unused variable _, we need to evaluate 5 and then bind it to *x*:

```
> ((λ (x) (+ x 3)) 5)
8
```

Let's use the same trick we used for *mybegin* to make this code look nicer.

```
(define mylet
  (λ (x f)
    (f x)))
```

```
> (mylet 5 (λ (x) (+ x 3)))
8
```

We can also work easily with larger examples.

```
> (mylet 5 (λ (x) (mylet x (λ (y) (+ x y)))))
10
```

1.3 The programmable semicolon

Why, then, bother with two names for the same function? After all, the definitions of *mybegin* and *mylet* are identical. Let's call these what they really are, which is $bind_{identity}$.

```
(define bind_identity
  (λ (ma sequel)
    (sequel ma))) ; ⟸ This is a mb.
```

With a definition for $bind_{identity}$, we nearly have a monad; we also need a function $unit_{identity}$. Loosely speaking, $unit_M$ is a function that brings a value into the world of a monad M in a natural way. The identity monad's world is simply the world of Scheme values, so the natural choice is the identity function.

```
(define unit_identity
  (λ (a)
    a)) ; ⟸ This is a ma.
```

Our examples are easily translated into the identity monad by replacing *mylet* and *mybegin* with $bind_{identity}$, and by bringing our Scheme values (trivially) into the identity monad with $unit_{identity}$.

```
> (bind_identity (unit_identity (printf "One\n"))
    (λ (_) (bind_identity (unit_identity (printf "Two\n"))
             (λ (_) (unit_identity (printf "Three\n"))))))
One
Two
Three
> (bind_identity (unit_identity 5)
    (λ (x) (unit_identity (+ x 3))))
8
> (bind_identity (unit_identity 5)
    (λ (x) (bind_identity (unit_identity x)
             (λ (y) (unit_identity (+ x y))))))
10
```

The identity monad by itself isn't terribly useful. After all, we can write these examples more concisely with **begin** and **let**. What we've done, though, is provide a hook into how we evaluate expressions in sequence; $bind_M$ is a programmable semicolon[2]. $bind_{identity}$ encodes a very basic notion of sequencing, "do this, then do that". The way we have structured the code that uses $bind_{identity}$ allows us to switch to more complex notions of sequencing simply by swapping $bind_{identity}$ for a differently-programmed semicolon.

2 The State Monad

Here is a predicate *even-length?* which takes a list *ls*, and then returns #t if the length of *ls* is even, and #f otherwise.

```
(define even-length?
  (λ (ls)
    (cond
      ((null? ls) #t)
      (else (not (even-length? (cdr ls)))))))

> (even-length? '(1 2 3 4))
#t
```

[2]See "Real World Haskell" by Bryan O'Sullivan, Don Stewart, and John Goerzen http://book.realworldhaskell.org/read/monads.html#id642960

Suppose we want to rewrite *even-length?* using store-passing style. We add the store as an argument s and give it the initial value #t. Each time we recur, we negate the value of s, so that we are left with the correct answer in s at the end of the computation.

```
(define even-length?_sps
  (λ (ls s)
    (cond
      [(null? ls) s]
      [else (even-length?_sps (cdr ls) (not s))])))
```

```
> (even-length?_sps '(1 2 3 4) #t)
#t
```

The *state* monad allows us to write programs that use state without the overhead of adding an extra argument like s to all of our functions. It accomplishes this without **set!** or its relatives, providing an illusion of a mutable variable with purely-functional code.

Here is *even-length?*$_{state}$, which uses the state monad to replace the extra argument to *even-length?*$_{sps}$.

```
(define even-length?_state
  (λ (ls)
    (cond
      ((null? ls) (unit_state '_))
      (else (bind_state
              (λ (s)
                `(_ . ,(not s)))
              (λ (_)
                (even-length?_state (cdr ls))))))))
```

```
> ((even-length?_state '(1 2 3 4)) #t)
(_ . #t)
```

This resulting value is unusual. Where *even-length?* and *even-length?*$_{sps}$ return a boolean value, *even-length?*$_{state}$ returns a pair whose *cdr* is the boolean value we expect, and whose *car* is the symbol _. Running a computation in the state monad always returns a pair of a natural value and final state of the computation. For *even-length?*$_{state}$, we only care about the final state, so we use _ throughout the program as a convention to indicate that the natural value is irrelevant.

More unusual than the resulting value are the two new functions that appear in this definition: *unit*$_{state}$ and *bind*$_{state}$. These functions comprise the state monad.

```
(define unit_state
  (λ (a)
    (λ (s) ; ⟸ This function is a ma.
      `(,a . ,s))))
```

unit$_{state}$ takes a natural value a and returns a trivial computation in the state monad. When passed a state s, this trivial computation returns a pair of a and s, both unchanged.

```
(define bind_state
  (λ (ma sequel)
    (λ (s) ; ⟸ This function is a mb.
      (let ((p (ma s)))
        (let ((â (car p)) (ŝ (cdr p)))
          (let ((mb (sequel â)))
            (mb ŝ)))))))
```

bind$_{state}$ composes two state monad computations into a single computation. This composition requires that any changes to the state made by the first computation be visible to the second.

To accomplish this, *bind*$_{state}$ passes an initial state s into the first computation ma, which returns a pair ($\hat{a}$. $\hat{s}$), the natural value and resulting state of running ma. Then, it passes $\hat{a}$ to the *sequel* function, which returns the second computation mb. With mb in hand, all that remains is to pass the intermediate state $\hat{s}$ to mb, yielding the result of the composed computation.

even-length?$_{state}$ doesn't use the full power these functions give us since it always ignores the natural value. Let's look at an example that uses both the state and the natural value. The task is to take a nested (any depth) list of integers and return as the natural value the list with all even numbers removed. The state of the computation will be a running tally of the even numbers that have been deleted, so that when the computation finishes, it will be the count of all the even numbers in the original list. We call this function *remberevens*X*countevens*. The cross X indicates that the function returns an eXtra value.

Before we move on to a monadic definition of *remberevens*X*countevens*, let's again look at a simple, direct-style definition. We start with a "driver" procedure, *remberevens*X*countevens*$_{2pass}$, that calls off to two helpers, *remberevens*$_{direct}$ and *countevens*$_{direct}$.

```
(define remberevensXcountevens_2pass
  (λ (l)
    `(,(remberevens_direct l) . ,(countevens_direct l))))

(define remberevens_direct
  (λ (l)
    (cond
      ((null? l) '())
      ((pair? (car l)) (cons (remberevens_direct (car l)) (remberevens_direct (cdr l))))
      ((or (null? (car l)) (odd? (car l))) (cons (car l) (remberevens_direct (cdr l))))
      (else (remberevens_direct (cdr l))))))

(define countevens_direct
  (λ (l)
    (cond
      ((null? l) 0)
      ((pair? (car l)) (+ (countevens_direct (car l)) (countevens_direct (cdr l))))
      ((or (null? (car l)) (odd? (car l))) (countevens_direct (cdr l)))
      (else (add1 (countevens_direct (cdr l)))))))

> (remberevensXcountevens_2pass '(2 3 (7 4 5 6) 8 (9) 2))
((3 (7 5) (9)) . 5)
```

The *rememberevens*X*countevens*$_{2pass}$ solution works, but is inefficient: it processes the list l twice. There is a well-known way to get the same answer, and yet process the list once, but the solution requires that we transform the code into continuation-passing style.

```
(define remberevensXcountevens_cps
  (λ (l k)
    (cond
      ((null? l) (k `(() . 0)))
      ((pair? (car l))
       (remberevensXcountevens_cps (car l)
         (λ (pa)
           (remberevensXcountevens_cps (cdr l)
             (λ (pd)
               (k `(,(cons (car pa) (car pd)) . ,(+ (cdr pa) (cdr pd)))))))))
      ((or (null? (car l)) (odd? (car l)))
       (remberevensXcountevens_cps (cdr l)
         (λ (p)
           (k `(,(cons (car l) (car p)) . ,(cdr p))))))
      (else (remberevensXcountevens_cps (cdr l)
              (λ (p) (k `(,(car p) . ,(add1 (cdr p))))))))))
```

```
> (remberevensXcountevens_cps '(2 3 (7 4 5 6) 8 (9) 2) (λ (p) p))
((3 (7 5) (9)) . 5)
```

Next we transform the direct-style *remberevens*$_{direct}$ into monadic style. The fourth clause is a tail call, so it remains unchanged. In the third clause, we take the nontail call (with simple arguments) and make it the first argument to *bind*$_{state}$.

```
(bind_state (remberevens_direct (cdr l)) ...)
```

The context around the nontail call goes into the "..." and we must have a variable to bind the natural value of the call to (*remberevens*$_{direct}$ (*cdr* l)), so let's call it d.

```
(bind_state (remberevens_direct (cdr l)) (λ (d) ...))
```

The (λ (d) ...) here is the *sequel* argument to bind, and since *bind*$_{state}$'s job is to thread the state from the first computation to the computation returned by the *sequel*, we need not worry at all about the state at this point. Next, if we have a simple expression (one without a recursive function call) like (*cons* (*car* l) d), then to monadify it, we use *unit*$_{state}$ around the simple expression.

```
(bind_state (remberevens_direct (cdr l))
  (λ (d) (unit_state (cons (car l) d))))
```

Consider the second clause. Here we have two nontail (recursive) calls (with simple arguments), so we have to sequence them.

```
(bind_state (remberevens_direct (car l))
  (λ (a) ...))
```

In the body of (λ (a) ...) we make the next call.

```
(bind_state (remberevens_direct (car l))
  (λ (a)
    (bind_state (remberevens_direct (cdr l))
      (λ (d) ...))))
```

Finally, we have unnested the recursive calls on both the *car* and the *cdr*, and all that's left is to (*cons a d*), which is simple. Once again we wrap the simple expression using *unit*.[3]

```
(bind_state (rememberevens_direct (car l))
  (λ (a)
    (bind_state (rememberevens_direct (cdr l))
      (λ (d) (unit_state (cons a d))))))
```

The first clause is simple: we simply pass '() to $unit_{state}$, and we have our result.

```
(define rememberevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((pair? (car l))
       (bind_state (rememberevens (car l))
         (λ (a)
           (bind_state (rememberevens (cdr l))
             (λ (d) (unit_state (cons a d)))))))
      ((or (null? (car l)) (odd? (car l)))
       (bind_state (rememberevens (cdr l))
         (λ (d) (unit_state (cons (car l) d)))))
      (else
       (rememberevens (cdr l))))))
```

Of course, all we've dealt with so far is *rememberevens*, and what we really wanted was *rememberevens*×*countevens*. It would seem that we've only done half of our job. However, the beauty of the state monad is that we are almost done. Let's change the name of the function to *rememberevens*×$countevens_{almost}$ and see just how far off we are.

---

[3] The nested calls to $bind_{state}$ could be made to look simpler with a macro $\mathbf{do}^*_{state}$, reminiscent of Haskell's **do** and Scheme's **let***.

```
(define-syntax do*_state
  (syntax-rules ()
    ((_ () body) body)
    ((_ ((a0 ma0) (a ma) ...) body)
     (bind_state ma0
       (λ (a0) (do*_state ((a ma) ...) body))))))

(do*_state ((a (rememberevens_direct (car l)))
            (d (rememberevens_direct (cdr l))))
  (unit_state (cons a d)))
```

```
(define remberevensXcountevens_almost
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((pair? (car l))
       (bind_state (remberevensXcountevens_almost (car l))
         (λ (a)
           (bind_state (remberevensXcountevens_almost (cdr l))
             (λ (d) (unit_state (cons a d)))))))
      ((or (null? (car l)) (odd? (car l)))
       (bind_state (remberevensXcountevens_almost (cdr l))
         (λ (d) (unit_state (cons (car l) d)))))
      (else
       (remberevensXcountevens_almost (cdr l))))))
```

First, what does (*remberevens*X*countevens*$_{almost}$ *l*) return? It returns a function that takes a state and returns a pair of values, the natural value that one might return from a call to (*remberevens*$_{direct}$ *l*) and the state, which is the number of even numbers that have been removed. Here is a test of *remberevens*X*countevens*$_{almost}$.

```
> ((remberevensXcountevens_almost '(2 3 (7 4 5 6) 8 (9) 2)) 0)
((3 (7 5) (9)) . 0)
```

What is 0 doing in the test? It is the initial value of the state *s*. What happens when the list of numbers is empty? Then, we return (*unit*$_{state}$ '()), which is a function (λ (*s*) ‘(() . ,*s*)), by substituting () for *a* in the body of *unit*$_{state}$. Then 0 is substituted for *s*, which yields the pair (() . 0).

But, our answer is *almost* correct, since the only part that is wrong is the count. When should we be counting? When we know we have an even number in (*car l*). So, let's look at that **else** clause again.

```
(remberevensXcountevens_almost (cdr l))
```

How can we revise this expression to fix the bug? This is a tail call, so we move the call into the body of a sequel.

```
(bind_state ...
  (λ (_) (remberevensXcountevens_almost (cdr l))))
```

Then we manufacture a state monad computation that modifies the state. In *even-length?*$_{state}$, (λ (*s*) ‘(_ . ,(*not s*))) is the computation we use to negate the state, which in that computation was a boolean value.[4] Here, we instead want to increment the state that is an integer. We don't care about the natural value of incrementing the state for the same reason we wouldn't care about the value of (**set!** *s* (*add1 s*)), so we'll again use _ for both the natural value and the variable that it will be bound to in the *sequel*.

```
(bind_state (λ (s) `(_ . ,(add1 s)))
  (λ (_) (remberevensXcountevens_almost (cdr l))))
```

Since the *s* coming into this computation is the current count, our computation yields the state (*add1 s*), and the **else** clause is finished. The code is now correct, so we drop the *almost* subscript from the name.

[4] Like the bodies of *unit*$_{state}$ and *bind*$_{state}$, state monad computations are of the form (λ (*s*) *body*) where *body* evaluates to a pair. The same principle applies to the *sequel*, which is of the form (λ (*a*) *mb*) where *mb* evaluates to a state monad computation.

```
(define remberevensXcountevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((pair? (car l))
       (bind_state (remberevensXcountevens (car l))
         (λ (a)
           (bind_state (remberevensXcountevens (cdr l))
             (λ (d) (unit_state (cons a d)))))))
      ((or (null? (car l)) (odd? (car l)))
       (bind_state (remberevensXcountevens (cdr l))
         (λ (d) (unit_state (cons (car l) d)))))
      (else
       (bind_state (λ (s) `(_ . ,(add1 s)))
         (λ (_) (remberevensXcountevens (cdr l))))))))

> ((remberevensXcountevens '(2 3 (7 4 5 6) 8 (9) 2)) 0)
((3 (7 5) (9)) . 5)
```

Let's think about the earlier definition in continuation-passing style. Both programs compute the correct answer, but they are doing so in very different ways. To show that this is the case, let's trace the execution of the *add1* and + *operators* as we run each version of the program. Here's what happens for *remberevensXcountevens*$_{cps}$:

```
> (remberevensXcountevens_cps '(2 3 (7 4 5 6) 8 (9) 2) (λ (p) p))
|(add1 0)
|1
|(add1 1)
|2
|(add1 0)
|1
|(+ 0 1)
|1
|(add1 1)
|2
|(+ 2 2)
|4
|(add1 4)
|5
((3 (7 5) (9)) . 5)
```

As we can see from the execution trace, *remberevensXcountevens*$_{cps}$ computes the number 5 by computing sub-answers for the various sub-lists in the input, then combining the sub-answers with +.

Let's look at a trace of the monadic version, *remberevens*X*countevens*:

```
> ((remberevensXcountevens '(2 3 (7 4 5 6) 8 (9) 2)) 0)
|(add1 0)
|1
|(add1 1)
|2
|(add1 2)
|3
|(add1 3)
|4
|(add1 4)
|5
((3 (7 5) (9)) . 5)
```

Now the results of calls to *add1* are following a predictable pattern, and + is never used at all! Instead of building up answers from sub-answers, as we see happening in the trace of $\mathit{remberevens}\mathsf{X}\mathit{countevens}_{cps}$, this version looks like we're incrementing a counter.

In fact, the computation that takes place is rather like what would have happened if we had created a global variable *counter*, initialized it to 0, and simply run (**set!** *counter* (*add1 counter*)) five times. But we do it all without having to use **set!**. Instead, the state monad provides us with the *illusion* of a mutable global variable. This is an extremely powerful idea. We can now write programs that provide a faithful simulation of effectful computation without actually performing any side effects—that is, we get the usual benefits of effectful computation without the usual drawbacks.

A final observation on the state monad is that the auxiliary function (λ (*s*) ‘(_ . ,(*add1 s*))), which contains no free variables, could have been given a global name, say incr_{state}.

```
(define incr_state (λ (s) `(_ . ,(add1 s))))
```

We might also recognize that it's common to apply arbitrary functions to the state rather than just *add1*, such as (λ (*s*) ‘(_ . ,(*not s*))) from $\mathit{even\text{-}length?}_{state}$. It is straightforward to define these both in terms of update_{state},

```
(define update_state
  (λ (f)
    (λ (s) `(_ . ,(f s)))))
(define incr_state (update_state add1))
(define negate_state (update_state not))
```

but then the relationship between the *ma* and *sequel* in a call to bind_{state}

$$\begin{array}{c} (\lambda\ (s)\ `(\_\ .\ ,(\mathit{add1}\ s))) \Longleftarrow \mathit{ma} \\ \Downarrow \\ (\lambda\ (\_)\ \ldots) \Longleftarrow \mathit{sequel} \end{array}$$

would not be as clear. The pure value, the symbol _, in the *car* of the pair returned when a state is passed to a *ma* is bound to the formal parameter, _, of the sequel. In addition to threading the state through the two computations, making this binding occur is how bind_{state} composes two computations.[5]

[5] We blithely use _, but it is not an odd or even integer. In Scheme, however, we have no real need to distinguish these types. We merely need to agree that we don't care about the fact that we are binding a useless value to a useless variable. Also, if we think about unit_{state} and bind_{state} as methods of some class C, we could imagine another class that inherits C and includes the incr_{state} method, but this is just packaging.

Exercise: In *rememberevens*X*countevens*, the increment takes place before the tail recursive call, but we are free to reorder these events. Implement this reordered-events variant by having the body of the *sequel* become the first argument to $bind_{state}$ and make the appropriate adjustments to the *sequel*. Is this new first argument to $bind_{state}$ a tail call?

Exercise: Define *remberevens*X*maxseqevens*, which removes all the evens, but while it does that, it also returns the length of the longest sequence of even numbers without an odd number. There are two obvious ways to implement this function; try to implement them both. Hint: Consider holding more than a single value in the state.

3 Deriving the State Monad

If we take the code for *remberevens*X*countevens* and replace the definitions of $unit_{state}$ and $bind_{state}$ by their definitions, opportunities for either (**let** ((*x e*)) *body*) or equivalently ((λ (*x*) *body*) *e*) exist for substituting *e* for *x* in *body*. If we know that *x* occurs in *body* just once, then these are correctness and efficiency (or better) preserving transformations. These transformations (all thirty-six) are in the appendix, worked out in detail, but, the result is the code in *store-passing style*, where a store is an argument passed in and out of every recursive function call. The resulting code is what we might have written had we not known of the *state* monad.

```
(define remberevensXcountevens_sps
  (λ (l s)
    (cond
      ((null? l) `(() . ,s))
      ((pair? (car l))
       (let ((p (remberevensXcountevens_sps (car l) s)))
         (let ((p̂ (remberevensXcountevens_sps (cdr l) (cdr p))))
           `(,(cons (car p) (car p̂)) . ,(cdr p̂)))))
      ((or (null? (car l)) (odd? (car l)))
       (let ((p (remberevensXcountevens_sps (cdr l) s)))
         `(,(cons (car l) (car p)) . ,(cdr p))))
      (else
       (let ((p (remberevensXcountevens_sps (cdr l) s)))
         `(,(car p) . ,(add1 (cdr p))))))))

> (remberevensXcountevens_sps '(2 3 (7 4 5 6) 8 (9) 2) 0)
((3 (7 5) (9)) . 5)
```

We can also start from $remberevens\mathsf{X}countevens_{sps}$ and derive $unit_{state}$ and $bind_{state}$, since each correctness-preserving transformation is invertible.

This ends the first monad lecture. In the second lecture, we will present various other monads and how one might use them.

Lecture 2: Other monads

4 Monads in a Nutshell

Each monad is a pair of functions, $unit_M$ and $bind_M$, that cooperate to do some rather interesting things. A particular $unit_M$, $bind_M$ pair is a monad if the following *monadic laws* hold:

- $(bind_M\ m\ unit_M) = m$
- $(bind_M\ (unit_M\ x)\ f) = (f\ x)$
- $(bind_M\ (bind_M\ m\ f)\ g) = (bind_M\ m\ (\lambda\ (x)\ (bind_M\ (f\ x)\ g)))$

Once we are at the point of developing our own monads, we will have to prove that the monadic laws hold for our proposed $unit_M$ and $bind_M$, but for now, we will only be dealing with known monads. If we wish to convince ourselves that a monad is truly a monad, we'll need to prove these laws.

5 Types and Shapes

Consider three types of values: *Pure* values, denoted by a and b; monadic expressions, denoted by ma and mb; and functions, denoted by $sequel_M$, that take a pure value a and return a monadic value mb. The $unit_M$ function is "shaped" something like a $sequel_M$, and $bind_M$ takes two arguments, a ma and a $sequel_M$, and returns a mb. We can therefore write down the *types* of $unit_M$ and $bind_M$ as follows.[6]

$unit_M : a \to ma$
$bind_M : ma \to (a \to mb) \to mb$; or $ma \to ((a \to mb) \to mb)$
$sequel_M = a \to mb$

Here, the last line simply tells us that the type $sequel_M$ is an abbreviation for the type $a \to mb$. The following two lines tell us the types of the expressions $unit_M$ and $bind_M$, respectively. We can read the colon, :, as "has the type".

From the monadic laws, we know that the expression $(bind_M\ m\ unit_M)$ is allowed, even though $bind_M$ seems to want a value of type $sequel_M$ as its second argument. Therefore, we know that $unit_M$ and a $sequel_M$ must have a similar shape. They both consume a pure value a and return either a ma or a mb. Furthermore, $(unit_M\ a)$ and $(bind_M\ ma\ sequel_M)$ both return the same shape, a ma or mb, respectively.

In this lecture we introduce several more monads by "instantiating", or replacing, the subscripted M and the m in ma and mb with a particular monad. In order for a particular choice of M to serve as a monad, we must define a particular pair of $unit_M$ and $bind_M$ that satisfies the monadic laws.

[6] Our use of $bind_M$ in Scheme expressions states that $bind_M$ takes two arguments, so a call would look like this: $(bind_M\ ma\ sequel)$. But the types below appear to state that $bind_M$ is curried so that a call to $bind_M$ would instead look like this: $((bind_M\ ma)\ sequel)$. This decision to show the types with $\to$ is to be consistent with the way the monad types are presented in the literature. In either case, we refer to the sequel as the second argument.

6 The List Monad

Here is the *list* monad.

```
(define unit_list
  (λ (a)
    `(,a))) ; ⟸ This list is a ma.

(define bind_list
  (λ (ma sequel)
    (mapcan sequel ma))) ; ⟸ This list is a mb.

(define mapcan
  (λ (f ls)
    (cond
      ((null? ls) '())
      (else (append (f (car ls)) (mapcan f (cdr ls)))))))
```

We know that a *ma* is a list of natural values, so each (*sequel a*) returns a list of natural values *mb*, thus the result of *mapcan* will also be a list of natural values.

We will find the auxiliaries $mzero^{list}$ and $mplus^{list}$ quite useful. In general, $mzero_M$ represents a computation with no answer in the monad M, and $mplus_M$ combines the answers from two computations. Not all monads have these notions; $unit_M$ and $bind_M$ are the only definitions common to all monads.

```
(define mzero^list '())

(define mplus^list append)
```

Consider this example from Jeff Newburn's tutorial. "The canonical example of using the List monad is for parsing ambiguous grammars. The example below shows [...] parsing data into hex values, decimal values, and words containing only alphanumeric characters. [...] hexadecimal digits overlap with both decimal digits and alphanumeric characters, leading to an ambiguous grammar. `"dead"` is both a valid hex value and a word, for example, and `"10"` is both a decimal value of 10 and a hex value of 16." (`"10"` is also an alphanumeric word.)

In the definition of *parse-c** below, we first create the three specialized parsers that take a pure tagged value and a new character. Then, we define the function that takes a tagged value and a list of characters. The same character is passed to these three defined parsers along with a tagged value. Each parser returns a *ma*, which are then formed into a list by combining the *ma*s together using $mplus^{list}$.

```
(define parse-c*
  (λ (a c*)
    (cond
      ((null? c*) (unit_list a))
      (else (bind_list (mplus^list
                         (parse-hex-digit a (car c*))
                         (parse-dec-digit a (car c*))
                         (parse-alphanumeric a (car c*)))
              (λ (a) (parse-c* a (cdr c*))))))))

(define char-hex?
  (λ (c)
    (or (char-numeric? c) (char≤? #\a c #\f))))

(define char-hex→integer/safe
  (λ (c)
    (− (char→integer c) (if (char-numeric? c) (char→integer #\0) (− (char→integer #\a) 10)))))
```

```
(define parse-hex-digit
  (λ (a c)
    (cond
      ((and (eq? (car a) 'hex-number) (char-hex? c))
       (unit_list `(hex-number . ,(+ (* (cdr a) 16) (char-hex→integer/safe c)))))
      (else mzero^list))))

(define parse-dec-digit
  (λ (a c)
    (cond
      ((and (eq? (car a) 'decimal-number) (char-numeric? c))
       (unit_list `(decimal-number . ,(+ (* (cdr a) 10) (- (char→integer c) 48)))))
      (else mzero^list))))

(define parse-alphanumeric
  (λ (a c)
    (cond
      ((and (eq? (car a) 'word-string) (or (char-alphabetic? c) (char-numeric? c)))
       (unit_list `(word-string . ,(string-append (cdr a) (string c)))))
      (else mzero^list))))
```

Below we produce a legal hex and alphanumeric string. Again, the hex string has been converted to the decimal number, 171.

```
> (bind_list (mplus^list
               (unit_list '(hex-number . 0))
               (unit_list '(decimal-number . 0))
               (unit_list '(word-string . "")))
     (λ (a) (parse-c* a (string→list "ab"))))
((hex-number . 171) (word-string . "ab"))
```

Next, we get a legal hex number, decimal number, and alphanumeric string.

```
> (bind_list (mplus^list
               (unit_list '(hex-number . 0))
               (unit_list '(decimal-number . 0))
               (unit_list '(word-string . "")))
     (λ (a) (parse-c* a (string→list "123"))))
((hex-number . 291) (decimal-number . 123) (word-string . "123"))
```

Of course, if we discover a special character, we fail by returning the empty list of answers.

```
> (bind_list (mplus^list
               (unit_list '(hex-number . 0))
               (unit_list '(decimal-number . 0))
               (unit_list '(word-string . "")))
     (λ (a) (parse-c* a (string→list "abc@x"))))
()
```

7 The Maybe Monad

Here is the *maybe* monad.

```
(define unit_maybe
  (λ (a)
    `(Just ,a))) ; ⟸ This is a ma.

(define bind_maybe
  (λ (ma sequel)
    (cond ; ⟸ This is a mb.
      ((eq? (car ma) 'Just)
       (let ((a (cadr ma)))
         (sequel a)))
      (else ma))))
```

A *ma* in the maybe monad is either a list of the form (*Just a*) where *a* is a natural value, or (*Nothing*). The *Just* tag means the computation was successful, while *Nothing* indicates failure.

If you have ever used Scheme's *assq*, then you know the ill-structured mess of always explicitly checking for failure. The maybe monad allows the programmer to think at a higher level when handling of failure is not relevant. Consider *new-assq*, which is like *assq*. Its job is to return (*Just a*) where *a* is the *cdr* of the first pair in p^* whose *car* matches *v*.

```
(define new-assq
  (λ (v p*)
    (cond
      ((null? p*) '(Nothing)) ; ⟸ (Nothing) is a ma representing failure
      ((eq? (caar p*) v) (unit_maybe (cdar p*)))
      (else (bind_maybe (new-assq v (cdr p*))
              (λ (a) (unit_maybe a)))))))
```

Since (*new-assq v* (*cdr* p^*)) is a tail call, we can rewrite *new-assq* relying on η reduction and the first monadic law, leading to

```
(define new-assq
  (λ (v p*)
    (cond
      ((null? p*) '(Nothing))
      ((eq? (caar p*) v) (unit_maybe (cdar p*)))
      (else (new-assq v (cdr p*))))))
```

All right-hand sides of each **cond**-clause must be *ma*s, of course. We see that they are since the only way to terminate is in the first two **cond**-clauses, and each is a *ma*. To see how we might use *new-assq*, we run the following test.

```
> (bind_maybe
    (let ((ma1 (new-assq 8 '((7 . 1) (9 . 3)))))
      (cond
        ((eq? (car ma1) 'Just) ma1)
        (else (let ((ma2 (new-assq 8 '((9 . 4) (6 . 5) (8 . 2) (7 . 3)))))
                ma2))))
    (λ (a) (new-assq a '((1 . 10) (2 . 20)))))
(Just 20)
```

We have to verify that the first argument to $bind_{maybe}$ is a *ma*. In either clause of the **cond** expression above, the result is a *ma*. Here we are looking up 8 in two different association lists. Since 8 is not in the first association list, *ma1* is (*Nothing*), so the first **cond** clause fails and we try looking up 8 in the other association list. This succeeds with (*Just* 2), so the pure variable *a* in the *sequel* gets bound to the pure value 2. We are then taking the pure value 2 and looking it up in a third association list, which returns (*Just* 20).

8 The Continuation Monad

Here is the *continuation* monad.

```
(define unit_cont
  (λ (a)
    (λ (k) ; ⟸ This function is a ma.
      (k a))))

(define bind_cont
  (λ (ma sequel)
    (λ (k) ; ⟸ This function is a mb
      (let ((k̂ (λ (a)
                 (let ((mb (sequel a)))
                   (mb k)))))
        (ma k̂)))))
```

If we monadify the definition of *rememberevens*X*countevens*$_{cps}$ using the *continuation* monad, then the definition of *rememberevens*X*countevens* becomes a single argument procedure.

```
(define rememberevensXcountevens
  (λ (l)
    (cond
      ((null? l) (unit_cont `(() . 0)))
      ((pair? (car l))
       (bind_cont (rememberevensXcountevens (car l))
         (λ (pa)
           (bind_cont (rememberevensXcountevens (cdr l))
             (λ (pd)
               (unit_cont `(,(cons (car pa) (car pd)) . ,(+ (cdr pa) (cdr pd)))))))))
      ((or (null? (car l)) (odd? (car l)))
       (bind_cont (rememberevensXcountevens (cdr l))
         (λ (p)
           (unit_cont `(,(cons (car l) (car p)) . ,(cdr p))))))
      (else (bind_cont (rememberevensXcountevens (cdr l))
              (λ (p)
                (unit_cont `(,(car p) . ,(add1 (cdr p))))))))))

> ((rememberevensXcountevens '(2 3 (7 4 5 6) 8 (9) 2)) (λ (p) p))
((3 (7 5) (9)) . 5)
```

This should be enough evidence that our code is in continuation-passing style without an explicit continuation being passed around. We could use a similar derivation that shows how to regain the earlier explicit CPS'd definition, just as we generated store-passing style in the first lecture. We leave that as a tedious exercise for the reader.

Notably, the *continuation* monad allows us to write programs that use something very similar to *call/cc*, which we will name *callcc*. Here is its definition.

```
(define callcc
  (λ (f)
    (λ (k)
      (let ((k-as-proc (λ (a) (λ (k_ignored) (k a)))))
        (let ((ma (f k-as-proc)))
          (ma k))))))
```

In *callcc* we package the incoming current continuation k in a function that will ignore the future current continuation and invoke the stored k. We call this function *k-as-proc*, pass it to f, and then pass the current continuation k to the resulting *ma*.

We can demonstrate *callcc* with a program that takes the same kind of argument as *remberevens* and *immediately* returns 0 if a zero is found, otherwise it forms the product of all the numbers in this list.

```
(define product
  (λ (ls exit)
    (cond
      ((null? ls) (unit_cont 1))
      ((pair? (car ls))
       (bind_cont (product (car ls) exit)
         (λ (a)
           (bind_cont (product (cdr ls) exit)
             (λ (d) (unit_cont (* a d)))))))
      ((null? (car ls)) (product (cdr ls) exit))
      ((zero? (car ls)) (exit 0))
      (else (bind_cont (product (cdr ls) exit)
              (λ (d) (unit_cont (* (car ls) d))))))))
```

The first test below handles the base case where 1 is returned without invoking *out*.

```
> ((callcc (λ (out) (product '() out)))
   (λ (x) x))
1
```

The next example corresponds to Scheme's (*add1* (*call/cc* (λ (*out*) (*product* '() *out*)))). We add one to the answer because, when the value is returned by the default continuation, *add1* is waiting.

```
> ((bind_cont (callcc (λ (out) (product '() out)))
     (λ (a) (unit_cont (add1 a))))
   (λ (x) x))
2
```

The third example shows how the Scheme expression (*add1* (*call/cc* (λ (*out*) (*product* '(5 0 5) *out*)))) would be translated monadically. Since *add1* is in the continuation, *out*, we end up adding one to zero.

```
> ((bind_cont (callcc (λ (out)
                        (product '(5 0 5) out)))
     (λ (a) (unit_cont (add1 a))))
   (λ (x) x))
1
```

Here, since there is no 0 in the list, we get the product of the numbers in the list being returned by invoking the default continuation.

```
> ((callcc
     (λ (out)
       (product '(2 3 (7 4 5 6) 8 (9) 2) out)))
   (λ (x) x))
725760
```

This last example behaves the same as this Scheme example.

```
(call/cc
  (λ (k0)
    ((car (call/cc (λ (k1)
                     (k0 (− (call/cc (λ (k2) (k1 `(,k2)))) 1)))))
      3)))
```

But, monadifying it is a bit tricky. The ((*car* □) 3) that is in the continuation of *k1* has to move to the first *sequel*, and similarly, the (*k0* (− □ 1)) has to move to the second *sequel*.

```
> ((callcc (λ (k0)
             (bind_cont (callcc (λ (k1)
                                  (bind_cont (callcc (λ (k2) (k1 `(,k2))))
                                    (λ (n) (k0 (− n 1))))))
               (λ (a) ((car a) 3)))))
   (λ (x) x))
2
```

9 The Exception Monad

The *maybe* monad neatly represents computations which may fail. However, since failure is represented by '(Nothing), we have no opportunity to distinguish different sorts of failures.

For example, a program using a network connection might fail because of a connection timeout or because of a failure to resolve a hostname. We would like to distinguish between these cases so that the program may retry the connection in the first case, or produce a meaningful error message in the second.

The *exception* monad is a straightforward extension of the *maybe* monad which gives this capability.

```
(define unit_exception
  (λ (a)
    `(Success ,a))) ; ⟸ This is a ma.

(define bind_exception
  (λ (ma sequel)
    (cond ; ⟸ This is a mb.
      ((eq? (car ma) 'Success)
       (let ((a (cadr ma)))
         (sequel a)))
      (else ma))))
```

Our example is from Jeff Newbern's (`http://www.haskell.org/all_about_monads/html/errormonad.html`) "All About Monads A comprehensive guide to the theory and practice of monadic programming in Haskell Version 1.1.0". To quote Newbern, "The example attempts to parse hexadecimal numbers and throws an exception if an invalid character is encountered." The construction of an exception *ma* in the **else** branch of *char-hex→integer* below indicates the throwing of an exception. The sequel does not get invoked and consequently the pure variable *a* does not get bound if the *ma* produced by *char-hex→integer* is an exception *ma*. Instead the exception *ma* is returned as the answer.

```
(define parse-hex-c*
  (λ (c* pos n)
    (cond
      ((null? c*) (unit_exception n))
      (else (bind_exception (char-hex→integer (car c*) pos)
              (λ (a)
                (parse-hex-c* (cdr c*) (+ pos 1) (+ (* n 16) a))))))))

(define char-hex?
  (λ (c)
    (or (char-numeric? c) (char≤? #\a c #\f))))

(define char-hex→integer/safe
  (λ (c)
    (- (char→integer c) (if (char-numeric? c) (char→integer #\0) (- (char→integer #\a) 10)))))

(define char-hex→integer
  (λ (c pos)
    (cond
      ((char-hex? c) (unit_exception (char-hex→integer/safe c)))
      (else `(Exception ,(format "At index ~s : bad char ~c" pos c))))))

> (parse-hex-c* (string→list "ab") 0 0)
(Success 171)

> (parse-hex-c* (string→list "a5bex21b") 0 0)
(Exception "At index 4 : bad char x")
```

Normally, the two 0s passed to *parse-hex-c** should be hidden from the *parse-hex-c** interface, and that would be easy with a recursively defined local function within *parse-hex-c** that initializes the two variables. Furthermore, it is possible to introduce *catch* with exception handlers and *throw* to hide the representation used by the monad. Each of these things improves the definition, but also makes it more difficult to see how the *exception* monad is working.

Exercise: add the functions for *catch* and *throw*.

Exercise: Another approach is to define a global function $run_{exception}$ which takes an *exception* monad computation and returns either the untagged successful result or the exception if one is raised.

10 The Writer Monad

Here is the *writer* monad.

```
(define unit_writer
  (λ (a)
    `(,a . ()))) ; ⟸ This is a ma.

(define bind_writer
  (λ (ma sequel)
    (let ((a (car ma))) ; ⟸ This is a mb.
      (let ((mb (sequel a)))
        (let ((b (car mb)))
          `(,b . ,(append (cdr ma) (cdr mb)))))))))
```

The *writer* monad is used for programs which must produce output as they are evaluated. Programs frequently use the *writer* monad to produce logs. In this example, *reciprocals* returns a list of reciprocals of the numbers in its input. If it encounters a 0, it appends an error to its log, and then proceeds with the rest of the computation.

```
(define reciprocals
  (λ (ls)
    (cond
      ((null? ls) (unit_writer '()))
      ((zero? (car ls))
       (bind_writer `(_ . ("Saw a 0"))
         (λ (_)
           (reciprocals (cdr ls)))))
      (else (bind_writer (reciprocals (cdr ls))
              (λ (d)
                (unit_writer (cons (/ 1 (car ls)) d))))))))r
```

```
> (reciprocals '(1 2 3 4))
((1 1/2 1/3 1/4) . ())
> (reciprocals '(1 2 3 0 4 0 0 5 6 7))
((1 1/2 1/3 1/4 1/5 1/6 1/7) . ("Saw a 0" "Saw a 0" "Saw a 0"))
```

The writer monad builds its output using a monoid (a pair of an abstract addition operator and an abstract zero value that acts *addition-like*. Among such monoid pairs are (+, 0), (∗, 1) (*append*, ()), (**and**, #t), and (**or**, #f). In fact, any values we associate with *mplus* and *mzero* must also have these properties. Our particular implementation of $unit_{writer}$ and $bind_{writer}$ uses (*append*, ()).

Exercise: Modify the definitions above so that the log is a single string, rather than a list of strings.

```
> (reciprocals '(1 2 3 4))
((1 1/2 1/3 1/4) . "")
> (reciprocals '(1 2 3 0 4 0 0 5 6 7))
((1 1/2 1/3 1/4 1/5 1/6 1/7) . "Saw a 0\nSaw a 0\nSaw a 0\n")
```

11 The Reader Monad

Here is the *reader* monad.

```
(define unit_reader
  (λ (a)
    (λ (v) ; ⟸ This function is a ma.
      a)))

(define bind_reader
  (λ (ma sequel)
    (λ (v) ; ⟸ This function is a mb.
      (let ((a (ma v)))
        (let ((mb (sequel a)))
          (mb v))))))
```

The *reader* monad is very similar to the *state* monad, but when we use it, we only *initialize* the state. The illusion of a mutable variable is replaced by the illusion of a global variable whose value can be accessed anywhere in the computation.

In practice, these variables are quite useful in programs that are parameterized by a file handle, network, or database connection. Such programs need constant access to the variable, but rarely need to change its value.

The example program *multbydepth* takes an arbitrarily-nested list of numbers and returns the list with each number multiplied by its depth within the list.

```
(define multbydepth
  (λ (ls)
    (cond
      ((null? ls) (unit_reader '()))
      ((pair? (car ls))
       (bind_reader ask_reader
         (λ (v)
           (let ((a (run_reader (multbydepth (car ls)) (add1 v))))
             (bind_reader (multbydepth (cdr ls))
               (λ (d)
                 (unit_reader (cons a d))))))))
      (else (bind_reader ask_reader
              (λ (v)
                (bind_reader (multbydepth (cdr ls))
                  (λ (d)
                    (unit_reader (cons (* (car ls) v) d))))))))))

> (run_reader (multbydepth '(1 2 3 4)) 1)
(1 2 3 4)
> (run_reader (multbydepth '(1 (2) 3 4)) 1)
(1 (4) 3 4)
> (run_reader (multbydepth '(1 ((2) 3) 4)) 1)
(1 ((6) 6) 4)
```

We use the variable of the *reader* monad to keep track of the current depth. When the depth increases, we run another *reader* monad computation with the variable initialized to one more than the current depth. Note that the result of running this computation is a pure value rather than a *ma*, so we use **let** to give it a name rather than using $bind_{reader}$.

For readability, we use the straightforward run_{reader} and ask_{reader} combinators to start a *reader* computation and to ask for the *reader* variable.

(**define** run_{reader}
  (λ (ma v)
    (ma v)))

(**define** ask_{reader}
  (λ (v) v))

12 Appendix : State-Passing Style Derivation

We want to actually maintain the illusion of a state in non-monadic functional Scheme. To do this, we will need to pass the state in and out of every recursive (nonsimple) call. We will derive the definition that would have been produced in the absence of $unit_{state}$ and $bind_{state}$. We start our complete thirty-six step solution.

```
(define remberevens×countevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((pair? (car l))
       (bind_state (remberevens×countevens (car l))
         (λ (a)
           (bind_state (remberevens×countevens (cdr l))
             (λ (d) (unit_state (cons a d)))))))
      ((or (null? (car l)) (odd? (car l)))
       (bind_state (remberevens×countevens (cdr l))
         (λ (d) (unit_state (cons (car l) d)))))
      (else
       (bind_state (λ (s) `(_ . ,(add1 s)))
         (λ (_) (remberevens×countevens (cdr l))))))))
```

Before we dive into a lengthy derivation, it is necessary to make two observations.

1. ((λ (*x*) *body*) *e*) is equivalent to (**let** ((*x e*)) *body*).
2. In (**let** ((*x e*)) *body*) it is legitimate to substitute *e* for *x* in *body* provided that no unwanted variable capture occurs, and of course, this works in both directions.

For example, ((*f x*) ((*g x*) (*g x*))) can be rewritten as (**let** ((*gx* (*g x*))) ((*f x*) (*gx gx*))) and vice versa.

These are the primary transformations we use in the derivation below. Furthermore, we have structured the derivation where no unwanted variable capture can occur. It is always easy to avoid such variable capture by carefully renaming some variables.

The definition below is fully expanded: there are neither *bind*s nor *unit*s. The notation we use is that we are replacing an arbitrary variable x by some expression e, which we write as $[e/x]$. Our first two steps are $[.../unit_{state}]$ and $[.../bind_{state}]$. We use "…" when an expression is large and when there is no ambiguity as to what should be substituted for the variable.

All of the uses of $[e/x]$ in this derivation are unambiguous, by design—shadowing of lexical variables will not be a concern. Each step can be tested and will produce the correct answer. This property insures that a typographical error does not persist through these transformations, only to be discovered when the end result fails.

1/2. [.../$unit_{state}$] and [.../$bind_{state}$].

```
(define remberevens×countevens
  (λ (l)
    (cond
      ((null? l) ((λ (a) (λ (s) `(,a . ,s))) '()))
      ((pair? (car l))
       ((λ (ma sequel)
          (λ (s)
            (let ((p (ma s)))
              (let ((new-a (car p)) (new-s (cdr p)))
                (let ((mb (sequel new-a)))
                  (mb new-s))))))
        (remberevens×countevens (car l))
        (λ (a)
          ((λ (ma sequel)
             (λ (s)
               (let ((p (ma s)))
                 (let ((new-a (car p)) (new-s (cdr p)))
                   (let ((mb (sequel new-a)))
                     (mb new-s))))))
           (remberevens×countevens (cdr l))
           (λ (d) ((λ (a) (λ (s) `(,a . ,s))) (cons a d)))))))
      ((or (null? (car l)) (odd? (car l)))
       ((λ (ma sequel)
          (λ (s)
           (let ((p (ma s)))
             (let ((new-a (car p)) (new-s (cdr p)))
               (let ((mb (sequel new-a)))
                 (mb new-s))))))
        (remberevens×countevens (cdr l))
        (λ (d) ((λ (a) (λ (s) `(,a . ,s))) (cons (car l) d)))))
      (else
       ((λ (ma sequel)
          (λ (s)
            (let ((p (ma s)))
              (let ((new-a (car p)) (new-s (cdr p)))
                (let ((mb (sequel new-a)))
                  (mb new-s))))))
        (λ (s) `(_ . ,(add1 s)))
        (λ (_) (remberevens×countevens (cdr l))))))))
```

We start on the fourth clause.

3. [(λ (*a*) (λ (*s*) ‘(,*a* . ,(*add1* *s*))))/*sequel*].

```
(else
  ((λ (ma)
     (λ (s)
       (let ((p (ma s)))
         (let ((new-a (car p)) (new-s (cdr p)))
           (let ((mb ((λ (a) (λ (s) `(,a . ,(add1 s))))
                      new-a)))
             (mb new-s))))))
   (remberevens×countevens (cdr l))))
```

4. [(*remberevens*×*countevens* (*cdr* *l*))/*ma*].

```
(else
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     (let ((new-a (car p)) (new-s (cdr p)))
       (let ((mb ((λ (a) (λ (s) `(,a . ,(add1 s))))
                  new-a)))
         (mb new-s))))))
```

5. [.../*a*].

```
(else
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     (let ((new-a (car p)) (new-s (cdr p)))
       (let ((mb (λ (s) `(,new-a . ,(add1 s)))))
         (mb new-s))))))
```

6/7. [(*car* *p*)/*new-a*] and [(*cdr* *p*)/*new-s*].

```
(else
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     (let ((mb (λ (s) `(,(car p) . ,(add1 s)))))
       (mb (cdr p))))))
```

8. [(λ (*s*) ‘(,(*car* *p*) . ,(*add1* *s*)))/*mb*].

```
(else
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     ((λ (s) `(,(car p) . ,(add1 s))) (cdr p)))))
```

Now, we finish the clause.

9. [(*cdr* *p*)/*s*].

```
(else
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     `(,(car p) . ,(add1 (cdr p))))))
```

For the third clause, we can do approximately the same set of reductions as in the fourth clause, except there will be a slight difference, because of the way the pair will get constructed, but it should be obvious. First we fill in a value for *sequel* as before.

10. [(λ (*d*) ((λ (*a*) (λ (*s*) ‘(,*a* . ,*s*))) (*cons a d*)))/*sequel*].

```
((or (null? (car l)) (odd? (car l)))
 ((λ (ma)
    (λ (s)
      (let ((p (ma s)))
        (let ((new-a (car p)) (new-s (cdr p)))
          (let ((mb ((λ (d)
                       ((λ (a)
                          (λ (s)
                            `(,a . ,s)))
                        (cons (car l) d)))
                     new-a)))
            (mb new-s))))))
  (rememberevens×countevens (cdr l))))
```

11. [(*rememberevens*×*countevens* (*cdr l*))/*ma*].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((rememberevens×countevens (cdr l)) s)))
     (let ((new-a (car p)) (new-s (cdr p)))
       (let ((mb ((λ (d)
                    ((λ (a)
                       (λ (s)
                         `(,a . ,s))) (cons (car l) d))) new-a)))
         (mb new-s))))))
```

12/13. [(*car p*)/*new-a*] and [(*cdr p*)/*new-d*].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((rememberevens×countevens (cdr l)) s)))
     (let ((mb ((λ (d)
                  ((λ (a)
                     (λ (s) `(,a . ,s)))
                   (cons (car l) d)))
                (car p))))
       (mb (cdr p))))))
```

14. [(*car p*)/*d*].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((rememberevens×countevens (cdr l)) s)))
     (let ((mb ((λ (a)
                  (λ (s) `(,a . ,s)))
                (cons (car l) (car p)))))
       (mb (cdr p))))))
```

15. [((*cons* (*car* *l*) (*car* *p*))/*a*)].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     (let ((mb (λ (s) `(,(cons (car l) (car p)) . ,s))))
       (mb (cdr p))))))
```

16. [(λ (*s*) ‘(,(*cons* (*car* *l*) (*car* *p*)) . ,*s*))/*mb*].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     ((λ (s) `(,(cons (car l) (car p)) . ,s)) (cdr p)))))
```

With this final step, we are done with the third clause.

17. [(*cdr* *p*)/*s*].

```
((or (null? (car l)) (odd? (car l)))
 (λ (s)
   (let ((p ((remberevens×countevens (cdr l)) s)))
     `(,(cons (car l) (car p)) . ,(cdr p)))))
```

To work through the second clause and maintain one's sanity, it is a good idea to rename some of the variables. We will add a hat on the variables in the inner code.

18. Rename variables.

```
((pair? (car l))
 ((λ (ma sequel)
    (λ (s)
       (let ((p (ma s)))
         (let ((new-a (car p)) (new-s (cdr p)))
           (let ((mb (sequel new-a)))
             (mb new-s))))))
  (remberevens×countevens (car l))
  (λ (â)
    ((λ (mâ sequel̂)
       (λ (ŝ)
          (let ((p̂ (mâ ŝ)))
            (let ((new-â (car p̂)) (new-ŝ (cdr p̂)))
              (let ((mb̂ (sequel̂ new-â)))
                (mb̂ new-ŝ))))))
     (remberevens×countevens (cdr l))
     (λ (d) ((λ (a) (λ (s) `(,a . ,s))) (cons â d)))))))
```

And so we begin.

19. [.../*sequel*].

```
((pair? (car l))
 ((λ (ma)
    (λ (s)
      (let ((p (ma s)))
        (let ((new-a (car p)) (new-s (cdr p)))
          (let ((mb ((λ (â)
                      ((λ (mâ sequel̂)
                            (λ (ŝ)
                              (let ((p̂ (mâ ŝ)))
                                (let ((new-â (car p̂))
                                     (new-ŝ (cdr p̂)))
                                  (let ((m̂b (sequel̂ new-â)))
                                    (m̂b new-ŝ))))))
                         (remberevensXcountevens (cdr l))
                         (λ (d)
                           ((λ (a)
                              (λ (s) `(,a . ,s)))
                            (cons â d)))))
                    new-a)))
            (mb new-s))))))
  (remberevensXcountevens (car l))))
```

20. [.../*ma*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevensXcountevens (car l)) s)))
     (let ((new-a (car p)) (new-s (cdr p)))
       (let ((mb ((λ (â)
                   ((λ (mâ sequel̂)
                       (λ (ŝ)
                         (let ((p̂ (mâ ŝ)))
                           (let ((new-â (car p̂))
                                (new-ŝ (cdr p̂)))
                             (let ((m̂b (sequel̂ new-â)))
                               (m̂b new-ŝ))))))
                     (remberevensXcountevens (cdr l))
                     (λ (d)
                       ((λ (a)
                          (λ (s) `(,a . ,s)))
                        (cons â d)))))
                 new-a)))
         (mb new-s))))))
```

21/22. [.../*new-a*] and [.../*new-s*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevensXcountevens (car l)) s)))
     (let ((mb ((λ (â)
                   ((λ (mâ sequel̂)
                      (λ (ŝ)
                        (let ((p̂ (mâ ŝ)))
                          (let ((new-â (car p̂))
                                (new-ŝ (cdr p̂)))
                            (let ((mb̂ (sequel̂ new-â)))
                              (mb̂ new-ŝ))))))
                    (remberevensXcountevens (cdr l))
                    (λ (d)
                      ((λ (a)
                         (λ (s) `(,a . ,s)))
                       (cons â d)))))
                 (car p))))
       (mb (cdr p))))))
```

23. [.../*mb*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevensXcountevens (car l)) s)))
     (((λ (â)
         ((λ (mâ sequel̂)
            (λ (ŝ)
              (let ((p̂ (mâ ŝ)))
                (let ((new-â (car p̂))
                      (new-ŝ (cdr p̂)))
                  (let ((mb̂ (sequel̂ new-â)))
                    (mb̂ new-ŝ))))))
          (remberevensXcountevens (cdr l))
          (λ (d)
            ((λ (a) (λ (s) `(,a . ,s)))
             (cons â d)))))
       (car p))
      (cdr p)))))
```

24. [(*car p*)/*â*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (((λ (mâ sequel̂)
          (λ (ŝ)
            (let ((p̂ (mâ ŝ)))
              (let ((new-â (car p̂))
                    (new-ŝ (cdr p̂)))
                (let ((mb̂ (sequel̂ new-â)))
                  (mb̂ new-ŝ))))))
        (remberevens×countevens (cdr l))
        (λ (d)
          ((λ (a) (λ (s) `(,a . ,s)))
           (cons (car p) d))))
      (cdr p)))))
```

25. [.../*sequel̂*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (((λ (mâ)
         (λ (ŝ)
           (let ((p̂ (mâ ŝ)))
             (let ((new-â (car p̂))
                   (new-ŝ (cdr p̂)))
               (let ((mb̂ ((λ (d)
                             ((λ (a) (λ (s) `(,a . ,s)))
                              (cons (car p) d)))
                           new-â)))
                 (mb̂ new-ŝ))))))
       (remberevens×countevens (cdr l)))
      (cdr p)))))
```

26. [(*remberevens×countevens* (*cdr l*))/*mâ*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     ((λ (ŝ)
        (let ((p̂ ((remberevens×countevens (cdr l)) ŝ)))
          (let ((new-â (car p̂))
                (new-ŝ (cdr p̂)))
            (let ((mb̂ ((λ (d)
                          ((λ (a) (λ (s) `(,a . ,s)))
                           (cons (car p) d)))
                        new-â)))
              (mb̂ new-ŝ)))))
      (cdr p)))))
```

27. [(*cdr p*)/$\hat{s}$].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       (let ((new-â (car p̂))
             (new-ŝ (cdr p̂)))
         (let ((mb̂ ((λ (d)
                      ((λ (a) (λ (s) `(,a . ,s)))
                       (cons (car p) d)))
                    new-â)))
           (mb̂ new-ŝ)))))))
```

28/29. [(*car* $\hat{p}$)/*new-*$\hat{a}$], and [(*cdr* $\hat{p}$)/*new-*$\hat{s}$].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       (let ((mb̂ ((λ (d)
                    ((λ (a) (λ (s) `(,a . ,s)))
                     (cons (car p) d)))
                  (car p̂))))
         (mb̂ (cdr p̂)))))))
```

30. [.../$m\hat{b}$].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       (((λ (d)
           ((λ (a)
              (λ (s) `(,a . ,s)))
            (cons (car p) d)))
         (car p̂))
        (cdr p̂))))))
```

31. [(*car* $\hat{p}$)/*d*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       (((λ (a) (λ (s) `(,a . ,s)))
         (cons (car p) (car p̂)))
        (cdr p̂))))))
```

32. [(*cons* (*car* *p*) (*car* $\hat{p}$))/*a*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       ((λ (s) `(,(cons (car p) (car p̂)) . ,s))
        (cdr p̂))))))
```

The next step finishes the second clause.

33. [(*cdr* $\hat{p}$)/*s*].

```
((pair? (car l))
 (λ (s)
   (let ((p ((remberevens×countevens (car l)) s)))
     (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
       `(,(cons (car p) (car p̂)) . ,(cdr p̂))))))
```

Now, we come to the first clause, and we revisit what we have thus far derived.

34. ['()/*a*].

```
(define remberevens×countevens
  (λ (l)
    (cond
      ((null? l)
       (λ (s)
         `(() . ,s)))
      ((pair? (car l))
       (λ (s)
         (let ((p ((remberevens×countevens (car l)) s)))
           (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
             `(,(cons (car p) (car p̂)) . ,(cdr p̂))))))
      ((or (null? (car l)) (odd? (car l)))
       (λ (s)
         (let ((p ((remberevens×countevens (cdr l)) s)))
           `(,(cons (car l) (car p)) . ,(cdr p)))))
      (else
       (λ (s)
         (let ((p ((remberevens×countevens (cdr l)) s)))
           `(,(car p) . ,(add1 (cdr p)))))))))
```

Next we can do an inverted staging of each of the clause's outer (λ (s) ...).

35. Inverted staging.

```
(define rememberevens×countevens
  (λ (l)
    (λ (s)
      (cond
        ((null? l) `(() . ,s))
        ((pair? (car l))
         (let ((p ((rememberevens×countevens (car l)) s)))
           (let ((p̂ ((rememberevens×countevens (cdr l)) (cdr p))))
             `(,(cons (car p) (car p̂)) . ,(cdr p̂)))))
        ((or (null? (car l)) (odd? (car l)))
         (let ((p ((rememberevens×countevens (cdr l)) s)))
           `(,(cons (car l) (car p)) . ,(cdr p))))
        (else
         (let ((p ((rememberevens×countevens (cdr l)) s)))
           `(,(car p) . ,(add1 (cdr p)))))))))
```

The last step is to uncurry our definition. Now instead of taking two arguments, one at a time, it takes them at the same time, and furthermore, we can see that the state enters and exits from all the calls to *rememberevens*×*countevens*.

36. Uncurry.

```
(define rememberevens×countevens
  (λ (l s)
    (cond
      ((null? l) `(() . ,s))
      ((pair? (car l))
       (let ((p (rememberevens×countevens (car l) s)))
         (let ((p̂ (rememberevens×countevens (cdr l) (cdr p))))
           `(,(cons (car p) (car p̂)) . ,(cdr p̂)))))
      ((or (null? (car l)) (odd? (car l)))
       (let ((p (rememberevens×countevens (cdr l) s)))
         `(,(cons (car l) (car p)) . ,(cdr p))))
      (else
       (let ((p (rememberevens×countevens (cdr l) s)))
         `(,(car p) . ,(add1 (cdr p))))))))
```

```
> (rememberevens×countevens '(2 3 (7 4 5 6) 8 (9) 2) 0)
((3 (7 5) (9)) . 5)
```

If we work the thirty-six steps backwards (and it is obvious that we can) from here, we will discover exactly where the *state* monad ($unit_{state}$ and $bind_{state}$) might have come from.

13 Conclusion

We have used the “Wadler” (`http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`) approach to explaining monads from “The Essence of Functional Programming”. But, there are differences. Wadler shows how to extend an interpreter whereas we show how to extend “The Little Schemer” programs; Wadler assumes a reading knowledge of Haskell whereas we assume knowledge of functions as values and a reading knowledge of Scheme. Notably, we do not assume a reading knowledge of types. In the final analysis, we believe our approach to be clearer for the novice and Wadler’s appraoch to be clearer for the more sophisticated reader.

14 Acknowledgements

We have had conversations over the years with various people about monads, but some stand out as important as we developed our own way of explaining them. We want to thank, in alphabetical order, Michael Adams, David Bender, Will Byrd, Matthias Felleisen, Robby Findler, Steve Ganz, Ron Garcia, Darius Jahandarie, Roshan James, Jiho Kim, Ed Kmett, Ramana Kumar, Oleg Kiselyov, Anurag Mendhekar, Joe Near, Chung-Chieh Shan, Amr Sabry, Jeremy Seik, Jonathan Sobel, Larisse Voufo, Mitch Wand, Jeremiah Willcock, and Brent Yorgey. We are grateful for the tutorial papers by Phil Wadler, the tutorial by Jeff Newbern, and the stunningly clear paper by Eugenio Moggi that we mention above. We also want to thank the C311/B521 support staff from 2009 through 2011, including Lindsey Kuper, Nilesh Mahajan, Melanie Dybvig, Rebecca Ingram, Dustin Dannenhauer, Claire Alvis, Alex Hearn, Zack Owens, Christine Brugger, and Karissa McKelvey.