A Schemer's View of Monads

Daniel P. Friedman

March 27, 2010

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 10, and here, it might help to have some familiarity with first-class continuations.

1 Monads in a Nutshell

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

- $(star_M\ unit_M) = Identity_M$
- $(o\ (star_M\ f)\ unit_M) = f$
- $(star_M\ (o\ (star_M\ f)\ g)) = (o\ (star_M\ f)\ (star_M\ g))$

where o is the *composition* function, defined as $(\lambda\ (f\ g)\ (\lambda\ (x)\ (f\ (g\ x))))$, that takes two functions and and composes them.

If we were in the business of handing out new monads to others, we would have to prove that the monadic laws hold for our proposed $unit_M$ and $star_M$, 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.

We need to write our code so that given two expressions, we can quickly observe which of the two expressions occurs first. We call this style of writing code *monadic style*. Understanding how to write in monadic style is easy if you can recognize when a function call is (and is not) a tail call with simple arguments. Once the definition has been put in monadic style, there is a mechanism to insert "operations" into the definition that gives the illusion of some effect. In this first lecture it will be the illusion of a settable variable. In the second lecture, we introduce other effects.

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

A definition in monadic style is over two functions: a *unit* and a *star*, which must form a monad. If we start with a definition of a recursive function f that looks like this:

(**define** f (λ (...) *body*))

then the same function monadified will look like this:

(**define** f
  (λ (*unit star*)
    (**if** (*monad? unit star*)
      (**letrec** ((f (λ (...) *body**)))
        f))))

Unfortunately, ensuring that *unit* and *star* form a monad requires more effort than writing a simple predicate. For now, we will assume that the programmer can trust *unit* and *star* to be a monad, simplifying our definition.

(**define** f
  (λ (*unit star*)
    (**letrec** ((f (λ (...) *body**)))
      f)))

We use *body** to denote that the *body* is in monadic style. But, if instead of passing in a specific *unit* and *star*, we **define** them globally with unique names like $unit_{state}$ and $star_{state}$, then *body** looks exactly like it was before except that now specific *unit*s and *star*s are used. Making this decision allows us to use **define** instead of **letrec** to support recursion.

(**define** f (λ (...) *body**))

Using global definitions is only one of several ways to package monads. We could have packaged these two functions in a cons pair, a vector, an inheritable class object, etc. Generally, to support the illusion of an effect, there will be one or more auxiliary functions that also work well with *unit* and *star*, and as we encounter them, we will point them out. These auxiliary functions return the same kind of values as the *invocation* of *unit*. But keep in mind that a monad is merely a pair of *unit* and *star* that satisfy the monadic laws.

2 Types and Shapes

Consider three types of values: *Pure* values, denoted by $a \in A$; expressions parameterized over A, denoted by $ma \in MA$[2]; and functions, denoted by $sequel \in Sequel$, that take a pure value a and return a monadic value $mb \in MB$ (pronounced "Mib"). The $unit_M$ function is "shaped" something like a value of type *Sequel*, but returns a MA instead of a MB, and $star_M$ takes two (curried) arguments, a sequel and a MA, and returns a MB. We can therefore write down the *types* of $unit_M$ and $star_M$ as follows.

$Sequel_M = A \to MB$
$unit_M : A \to MA$
$star_M : Sequel_M \to MA \to MB$; or $Sequel_M \to (MA \to MB)$

Here, the first 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 $star_M$, respectively. We can read the colon, :, as "has the type".

From the monadic laws, we know that the expression ($star_M$ $unit_M$) is allowed, even though $star_M$ seems to want a value of type *Sequel* as its first 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, respectively. Furthermore, ($unit_M$ a) and (($star$ $sequel_M$) ma) both return the same shape, a MA or MB, respectively.

[2] A MA is the person who gave birth to you. In this first lecture, we are only looking at the *state* MA.

3 The State Monad

Here is the *state* monad, so called because it creates an illusion of a single variable that we can mutate.

```
(define unit_state ; A → MA
  (λ (a)
    (λ (s) ; ⟸ This function is a MA.
      `(,a . ,s))))

(define star_state ; Sequel → MA → MB or Sequel → (MA → MB)
  (λ (sequel)
    (λ (ma)
      (λ (s) ; ⟸ This function is a MB.
        (let ((p (ma s)))
          (let ((new-a (car p)) (new-s (cdr p)))
            (let ((mb (sequel new-a)))
              (mb new-s))))))))
```

Let's analyze these definitions just a bit.[3] $unit_{state}$ takes a pure value $a \in A$ and returns a MA, a function that expects a state $s \in S$. When the MA gets a state, a pair is returned. The pair contains a pure value in its *car* and an *extra* value in its *cdr*. The pure value is passed to a sequel yielding a MB, and the extra value is passed to the MB.

Pure values will be in the *car* and extra values will be in the *cdr* of the pair returned from an invocation of a *state* MA or *state* MB. This reminds us that these structures always have a pure value, which can sometimes be useless. What's in the *cdr* is an extra value that supports the various illusions.

We can observe that $star_{state}$ takes two (curried) values, a function *Sequel* and a MA, which for states happens to be a function. Consider (λ (*ma*) *body*). Then *body* is a MB. But, in the definition of $star_{state}$, *ma* is invoked before *sequel*.[4] We can let pair p be the result of applying *ma* to some s. So, p's *car* is some pure *new-a*, which is passed to *sequel*. The result of applying *sequel* to the value *new-a* is *mb*, which as we know expects a state, and that state is in p's *cdr*. Thus, we can say that a *ma* enters with one state and a $mb \in MB$ exits with a perhaps different state.

The type information of monads tells us how we can use *unit* and *star* to define functions in *monadic* style. So, now let's look at an example. The problem is to take a nested (any depth) list of integers and return a pair of values. The first item in the pair is the same list, except that the even numbers have been removed, and the second item in the pair is the count of even numbers that have been deleted. 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 look at a simple, direct-style definition. We start with a "driver" procedure, $remberevens\mathsf{X}countevens_{2pass}$, that calls off to two helpers, $remberevens_{pure}$ and $countevens_{pure}$.

```
(define remberevensXcountevens_2pass
  (λ (l) `(,(remberevens_pure l) . ,(countevens_pure l))))

(define remberevens_pure
  (λ (l)
    (cond
      ((null? l) '())
      ((list?? (car l)) (cons (remberevens_pure (car l)) (remberevens_pure (cdr l))))
      ((odd? (car l)) (cons (car l) (remberevens_pure (cdr l))))
      (else (remberevens_pure (cdr l))))))
```

[3] The definitions of the various *unit* and *star* operators are more verbose than is necessary. We do this for pedagogical reasons. Feel free to make the obvious improvements as they are noticed.

[4] We define *bind* (spelled >>= in Haskell), which takes its two arguments $ma \in MA$ and $sequel \in Sequel$, where the argument order is the order of when things happen: (**define** $bind_{state}$ (λ (*ma sequel*) (($star_{state}$ *sequel*) *ma*))).

```
(define countevens_pure
  (λ (l)
    (cond
      ((null? l) 0)
      ((list?? (car l)) (+ (countevens_pure (car l)) (countevens_pure (cdr l))))
      ((odd? (car l)) (countevens_pure (cdr l)))
      (else (add1 (countevens_pure (cdr l)))))))

(define list??
  (λ (x)
    (or (null? l) (pair? l))))

> (rememberevens×countevens_2pass '(2 3 (7 4 5 6) 8 (9) 2))
((3 (7 5) (9)) . 5)
```

The *remberevens*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)))
      ((list?? (car l))
       (remberevensXcountevens_cps (car l)
         (λ (pa)
           (remberevensXcountevens_cps (cdr l)
             (λ (pd)
               (k `(,(cons (car pa) (car pd)) . ,(+ (cdr pa) (cdr pd)))))))))
      ((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*$_{pure}$ 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 second (curried) argument to *star*$_{state}$.

```
((star_state ...)
 (remberevens_pure (cdr l)))
```

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

```
((star_state (λ (d) ...))
 (remberevens_pure (cdr l)))
```

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.

```
((star_state (λ (d) (unit_state (cons (car l) d))))
 (remberevens_pure (cdr l)))
```

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

```
((star_state (λ (a) ...))
 (rememberevens_pure (car l)))
```

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

```
((star_state (λ (a)
               ((star_state (λ (d) ...))
                (remberevens_pure (cdr l)))))
 (remberevens_pure (car l)))
```

Finally, we have processed the recursive calls on both the *car* and the *cdr*, and we have only to (*cons a d*), which is simple. Once again we wrap the simple expression using *unit*.[5]

```
((star_state (λ (a)
               ((star_state (λ (d) (unit_state (cons a d))))
                (remberevens_pure (cdr l)))))
 (remberevens_pure (car l)))
```

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

```
(define remberevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((list?? (car l))
       ((star_state (λ (a)
                      ((star_state (λ (d) (unit_state (cons a d))))
                       (remberevens (cdr l)))))
        (remberevens (car l))))
      ((odd? (car l))
       ((star_state (λ (d) (unit_state (cons (car l) d))))
        (remberevens (cdr l))))
      (else
       (remberevens (cdr l))))))
```

[5] The nested stars could be made to look simpler with *bind*, as mentioned in an earlier footnote.

```
(bind (remberevens (car l))
  (λ (a)
    (bind (remberevens (cdr l))
      (λ (d) (unit_state (cons a d))))))
```

or 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)
     ((star_state (λ (a0) (do*_state ((a ma) ...) body)))
      ma0))))

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

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

```
(define rememberevensXcountevens_almost
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((list?? (car l))
       ((star_state (λ (a)
                      ((star_state (λ (d) (unit_state (cons a d))))
                       (rememberevensXcountevens_almost (cdr l)))))
        (rememberevensXcountevens_almost (car l))))
      ((odd? (car l))
       ((star_state (λ (d) (unit_state (cons (car l) d))))
        (rememberevensXcountevens_almost (cdr l))))
      (else
       (rememberevensXcountevens_almost (cdr l))))))
```

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

```
> ((rememberevensXcountevens_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*. 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.

```
(rememberevensXcountevens_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.

```
((star_state (λ (_)
               (rememberevensXcountevens_almost (cdr l))))
 ...)
```

and then manufacture a *MA* that can, through star_{state}, give the illusion of an effect. Since our *MA*s for the *state* monad look like (λ (*s*) ‘(,*a* . ,$\hat{s}$)), that is what we must use, and since we don't care what value will get bound to _, we might as well have the pure value be the symbol _, leading to

```
((star_state (λ (_) (rememberevensXcountevens_almost (cdr l))))
 (λ (s) `(_ . ŝ)))
```

All that is left is to decide what we want $\hat{s}$ to be. Since the *s* coming into this *MA* is the current count, then we can have $\hat{s}$ become (*add1 s*), leading us to complete the **else** clause.[6]

```
((star_state (λ (_) (rememberevensXcountevens_almost (cdr l))))
 (λ (s) `(_ . ,(add1 s))))
```

[6] In principle, the *state MA* is any expression that evaluates to a function, such that when the function gets an argument the body of the function evaluates to any pair. But, very little is lost thinking about the *MA* as a pattern instead of as something computed, which is why we are using quasiquotation. The same principle applies to the *Sequel*.

The code is now correct, so we drop the *almost* subscript from the name.

```
(define rememberevensXcountevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((list?? (car l))
       ((star_state (λ (a)
                      ((star_state (λ (d) (unit_state (cons a d))))
                       (rememberevensXcountevens (cdr l)))))
        (rememberevensXcountevens (car l))))
      ((odd? (car l))
       ((star_state (λ (d) (unit_state (cons (car l) d))))
        (rememberevensXcountevens (cdr l))))
      (else
       ((star_state (λ (_) (rememberevensXcountevens (cdr l))))
        (λ (s) `(_ . ,(add1 s))))))))
```

```
> ((rememberevensXcountevens '(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 in *rememberevens*X*countevens*$_{cps}$:

```
> (rememberevensXcountevens_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, *rememberevens*X*countevens*$_{cps}$ computes the number 5 by computing sub-answers for the various sub-lists in the input, then combining the sub-answers with +.

By contrast, let's look at a trace of the monadic version, *rememberevens*X*countevens*:

```
> ((rememberevensXcountevens '(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 *rememberevens*X$countevens_{cps}$, 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 settable global variable. This is an extremely powerful idea. We can now write programs that provide a faithful simulation of effectful computation, but 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*))))

but then the relationship between a *sequel* and its *ma*

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

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. Making this binding occur is one of the jobs of $star_{state}$.[7]

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 second (curried) argument to $star_{state}$ and make the appropriate adjustments to the sequel. Is this new second (curried) argument to $star_{state}$ a tail call?

Exercise: Define *rememberevens*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 the state holding more than a single value.

[7] 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 $star_{state}$ 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.

4 Deriving the State Monad

If we take the code for *rememberevens*X*countevens* and replace the definitions of $unit_{state}$ and $star_{state}$ 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 *state-passing style*, where a state 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 rememberevensXcountevens_sps
  (λ (l s)
    (cond
      ((null? l) `(() . ,s))
      ((list?? (car l))
       (let ((p (rememberevensXcountevens_sps (car l) s)))
         (let ((p̂ (rememberevensXcountevens_sps (cdr l) (cdr p))))
           `(,(cons (car p) (car p̂)) . ,(cdr p̂)))))
      ((odd? (car l))
       (let ((p (rememberevensXcountevens_sps (cdr l) s)))
         `(,(cons (car l) (car p)) . ,(cdr p))))
      (else
       (let ((p (rememberevensXcountevens_sps (cdr l) s)))
         `(,(car p) . ,(add1 (cdr p))))))))
```

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

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

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

Lecture 2: Other monads

In the second lecture we introduce several more monads. In order to define a monad, we must define two functions that work well together, that is, the monad must be certified.

5 The Maybe Monad

Here is the *maybe* monad.

```
(define unit_maybe
  (λ (a)
    `(,a . _))) ; ⟸ This MA gets its type from the type of a.
```

```
(define star_maybe
  (λ (sequel)
    (λ (ma)
      (cond ; ⟸ This is a MB.
        ((eq? (cdr ma) '_)
         (let ((a (car ma)))
           (sequel a)))
        (else (let ((mb ma))
                mb))))))
```

The tag _ in the *cdr* indicates that the pure value is in the *car* just as in the *state* monad. We immediately see that there are what appear to be extraneous aspects to this monad. If you recall, in the *state* monad everything was self contained; here however, things are not so clean, but since we are only concerned with $unit_{maybe}$ in the first two certification equations, and since there is an _ symbol that is used in $unit_{maybe}$ and a dispatch for the _ symbol in $star_{maybe}$, it is at least possible that the first two certification equations hold.

If you have ever used Scheme's *assq*, then you know what an ill-structured mess it is to always have to check 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 a *MA* (a pair) whose *car* is the *cdr* of the first pair in p^* whose *car* matches v.

```
(define new-assq
  (λ (v p*)
    (cond
      ((null? p*) '(_ . fail)) ; ⟸ (_ . fail) is a MA
      ((eq? (caar p*) v) (unit_maybe (cdar p*)))
      (else ((star_maybe (λ (a) (unit_maybe a)))
             (new-assq v (cdr p*)))))))
```

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

```
(define new-assq
  (λ (v p*)
    (cond
      ((null? p*) '(_ . fail))
      ((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, and they are since the only way to terminate is in the first two **cond**-clauses, and each is a *MA*. (Because (_ . fail)'s *cdr* is the symbol fail, it cannot be confused with _.) To see how we might use *new-assq*, we run the following test.

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

We have to verify that the second (curried) argument to $star_{maybe}$ 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. We are then taking the pure value 2 and looking it up in a third association list. This returns the pair (20 . _). In the **cond**-clause when we fail, we try the other *MA*, but in the case where we succeed, we use the one that succeeded. The pure variable *a* will get bound to the pure value 2. The downside of this definition is that the first two calls to *new-assq* will get evaluated, but that is only because we are not in a language like Haskell where the arguments are passed by need. If we wanted to get the benefits of Haskell, we would redefine the second *MA* to be a thunk.

```
> ((star_maybe (λ (a) (new-assq a '((1 . 10) (2 . 20)))))
   ((λ (Ma1 Ma2)
      (cond
        ((eq? (cdr Ma1) '_) Ma1)
        (else (Ma2))))
    (new-assq 8 '((7 . 1) (9 . 3)))
    (λ () (new-assq 8 '((9 . 4) (6 . 5) (8 . 2) (7 . 3))))))
(20 . _)
```

and still be convinced that the (curried) second argument would evaluate to a *MA*. Structurally, we could have chosen #f instead of (_ . fail) and appropriately revised the **cond**-clauses, however, because we show the *exception* monad next, we wanted to stay with the _ representations.

Exercise: Revise the *Maybe Monad* where it is assumed that every *MA* is a thunk.

6 The Exception Monad

Here is the exception monad, where again the pure value is in the *car*, but this time an exception, a string, is in the *cdr*, though any value other than the symbol _ would suffice.

```
(define unit_exception
  (λ (a)
    `(,a . _))) ; ⟸ This MA gets its type from the type of a.

(define star_exception
  (λ (sequel)
    (λ (ma)
      (cond ; ⟸ This is a MB.
        ((eq? (cdr ma) '_)
         (let ((a (car ma)))
           (sequel a)))
        (else (let ((mb ma))
                mb))))))
```

The definitions of $unit_{exception}$ ($star_{exception}$) and $unit_{maybe}$ (respectively $star_{maybe}$) are identical. 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 ((star_exception (λ (a)
                              (parse-hex-c* (cdr c*) (+ pos 1) (+ (* n 16) a))))
             (char-hex→integer (car c*) pos))))))

(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 `(_ . ,(format "At index ~s : bad char ~c" pos c))))))
```

Of course, the beauty of *parse-hex-c** is that if you think *purely*, there is nothing in the definition of *parse-hex-c** to indicate that anything might lead to an exception.

```
> (parse-hex-c* (string→list "ab") 0 0)
(171 . _)

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

Normally, the two 0's 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 might have been wiser to introduce *catch* with handlers and *throw* to hide representation, etc. But the reality is, each of these things improves various aspects of the definition, but it also would make it more difficult to understand the essence of the exception monad, which is the goal of this section.

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

Exercise: Another approach would be to define a global function $run_{exception}$ that that would act as the user interface and its job would be to invoke *parse-hex-c** and then return a single value, instead of a dotted pair with a useless _.

The next monad is the *writer* monad.

7 The Writer Monad

Here is the *writer* monad.

```
(define unit_writer
  (λ (a)
    `(,a . ,mzero^list))) ; ⟸ This pair is a MA.

(define star_writer
  (λ (sequel)
    (λ (ma)
      (let ((a (car ma))) ; ⟸ This is a MB.
        (let ((mb (sequel a)))
          (let ((new-b (car mb)))
            `(,new-b . ,(mplus^list (cdr ma) (cdr mb)))))))))
```

We need the auxiliaries $mzero^{list}$ and $mplus^{list}$.

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

(define mplus^list append)
```

We define *rememberevens*X*evens*, which takes the same argument as *rememberevens*X*countevens* and returns a pair that differs only in the *cdr*: instead of returning a count, it returns a list of the even numbers in the order that the even numbers were removed.

```
(define remberevensXevens
  (λ (l)
    (cond
      ((null? l) (unit_writer '()))
      ((list?? (car l))
       ((star_writer (λ (a)
                       ((star_writer (λ (d) (unit_writer (cons a d))))
                        (remberevensXevens (cdr l)))))
        (remberevensXevens (car l))))
      ((odd? (car l))
       ((star_writer (λ (d) (unit_writer (cons (car l) d))))
        (remberevensXevens (cdr l))))
      (else
       ((star_writer (λ (_) (remberevensXevens (cdr l))))
        `(_ . (,(car l))))))))
```

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

This is structurally similar to the *exception* monad, except we are building up our results 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 have these properties. The pair we implemented uses (*append*, ()).
Exercise: Produce an answer with the *cdr* reversed, but use the same test program and same definition of *remberevens*X*evens*. Hint: Solve it by redefining one global variable.

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

The next monad is the *list* monad.

8 The List Monad

Here is the *list* monad.

```
(define unit_list
  (λ (a)
    `(,a . ()))) ; ⟸ This pair is a MA.

(define star_list
  (λ (sequel)
    (λ (ma)
      (cond ; ⟸ This is a MB
        ((eq? (car ma) '_) '(_ . _))
        (else
         (let ((mb (sequel (car ma))))
           (let ((extra (append (cdr mb) (mapcan sequel (cdr ma)))))
             `(,(car mb) . ,extra))))))))

(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 pures, so each (*sequel a*) returns a *MB*, thus the result of *mapcan* will be a list of pures.

Consider this example (`http://www.haskell.org/all_about_monads/html/listmonad.html`) from Jeff Newburn's tutorial. "The canonical example of using the List monad is for parsing ambiguous grammars. The example below shows just a small example of parsing data into hex values, decimal values, and words containing only alphanumeric characters. Note that hexadecimal digits overlap 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 pure tagged value and a list of characters. The same character is passed to these three defined parsers along with a pure tagged value. Each returns a *MA*, which are then formed into a list by appending the *MA*s together using $mplus^{list}$.

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

```
(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.

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

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

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

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

This ends the discussion of the *list* monad. The next monad is the *environment* monad.

9 The Environment Monad

Here is the environment monad.

```
(define unit_environment
  (λ (a)
    (λ (env)
      (let ((ma a))
        ma))))

(define star_environment
  (λ (sequel)
    (λ (ma)
      (λ (env)
        (let ((a (ma env)))
          (let ((mb (sequel a)))
            (mb env)))))))

(define multDepth
  (λ (ls)
      (cond
        ((null? ls) (unit_environment '()))
        ((list?? ls)
         ((star_environment
            (λ (a)
              ((star_environment
                 (λ (d)
                   (unit_environment (cons a d))))
               (multDepth (cdr ls)))))
          (extend-context (multDepth (car ls)))))
        (else ((star_environment
                 (λ (a)
                   ((star_environment
                      (λ (d)
                        (unit_environment (* a d))))
                    (get-context))))
               (unit_environment ls))))))
```

This paragraph is out of date.

The *reader* monad is in effect the *state* monad, but when we use it, we only *initialize* the state, so if we think about our *almost* attempt that failed to increment the state, that would be an example of the *reader* monad. But, if the initial value had been say an association list of interesting global information, then every time we need that global information it would be accessible. The single settable variable illusion has disappeared and has been replaced by the illusion of a global variable whose value can be accessed. For example, we might have the following expression.

```
((star_reader (λ (assoc-list)
                (let ((animal (cdr (assq 'animal assoc-list))))
                  (if (or (eq? animal 'cat) (eq? animal 'dog))
                    (unit_reader 'domestic)
                    (unit_reader 'wild)))))
 (λ (s) (let ((a s)) `(,a . ,s))))
```

By making the association list s be treated as a pure value, a, in the MA, (λ (s) (**let** ((a s)) ‘(,a . ,s))), we can access that association list within the body of the sequel. How mutations to the state are denied is a matter of either self-discipline or externally-imposed discipline. Of course, the big advantage is that as far as the programmer is concerned, they never have to know anything about the association list unless it is needed, and most importantly, it is not passed around as an extra argument to all function calls, when in fact, it is only occasionally needed.

This ends our discussion of the *reader* monad. Next, we define programs that use an operator like Scheme's *call/cc*. It is nearly the same as Scheme's, but not quite, as we will soon discover.

10 The Continuation Monad

Here is the *continuation* monad.

```
(define unit_continuation
  (λ (a)
    (λ (k) ; ⟸ This function is a MA.
      (k a))))

(define star_continuation
  (λ (sequel)
    (λ (ma)
      (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 *remberevens*X*countevens*$_{cps}$ using the *continuation* monad, then the definition of *remberevens*X*countevens* becomes a single argument procedure.

```
(define remberevensXcountevens
  (λ (l)
    (cond
      ((null? l) (unit_continuation `(() . 0)))
      ((list?? (car l))
       ((star_continuation (λ (pa)
                              ((star_continuation
                                 (λ (pd)
                                   (unit_continuation `(,(cons (car pa) (car pd)) . ,(+ (cdr pa) (cdr pd))))))
                               (remberevensXcountevens (cdr l)))))
        (remberevensXcountevens (car l))))
      ((odd? (car l))
       ((star_continuation (λ (p)
                              (unit_continuation `(,(cons (car l) (car p)) . ,(cdr p)))))
        (remberevensXcountevens (cdr l))))
      (else ((star_continuation (λ (p)
                                   (unit_continuation `(,(car p) . ,(add1 (cdr p))))))
             (remberevensXcountevens (cdr l)))))))
```

```
> ((remberevensXcountevens '(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.

We would be done with the discussion of the *continuation* monad except that one of the great things about this monad is that the *continuation* monad allows us to write programs that use something very similar to *call/cc*, which we call *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 the definition of *callcc* we package the current continuation k to ignore the future current continuation and invoke the current stored one. That is what gets bound to *k-as-proc*. We pass that packaged continuation to f, which returns a *MA*, which is then passed the current k that we entered with. We demonstrate this with a program that takes the same kind of argument as *remberevens* and if a zero is found, the result is 0, otherwise it forms the product of all the numbers in this list. For the fun of it, we added some code to process after we exited to make sure that it did not happen. Had we not cared about the (*sub1* (*exit* 0)), which is the same as (*exit* 0), we might not have been convinced that we indeed have considered every facet of *call/cc*. This works because the task of monadifying (*sub1* (*exit* 0)) is to recognize that (*exit* 0) is a nontail call and so the call to *sub1* must be in the sequel, where it will be ignored.

```
(define product
  (λ (ls exit)
    (cond
      ((null? ls) (unit_continuation 1))
      ((list?? (car ls))
       ((star_continuation (λ (a)
                             ((star_continuation (λ (d) (unit_continuation (* a d))))
                              (product (cdr ls) exit))))
        (product (car ls) exit)))
      ((zero? (car ls)) ((star_continuation (λ (__) (unit_continuation (sub1 __))))
                         (exit 0)))
      (else ((star_continuation (λ (d) (unit_continuation (* (car ls) d))))
             (product (cdr ls) exit))))))
```

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.

```
> (((star_continuation (λ (a) (unit_continuation (add1 a))))
    (callcc (λ (out) (product '() out))))
   (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.

```
> (((star_continuation (λ (a) (unit_continuation (add1 a))))
    (callcc
      (λ (out)
        (product '(5 0 5) out))))
   (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 simple 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)
       ((star_continuation (λ (a) ((car a) 3)))
        (callcc
          (λ (k1)
            ((star_continuation (λ (n) (k0 (- n 1))))
             (callcc (λ (k2) (k1 `(,k2))))))))))
   (λ (x) x))
2
```

The next monad is the *identity* monad.

11 The Identity Monad

Here is the *identity* monad.

```
(define unit_identity
  (λ (a)
    (let ((ma a))
      ma)))

(define star_identity
  (λ (sequel)
    (λ (ma)
      (let ((a ma)) ; ⟸ This is a MB.
        (sequel a)))))
```

Consider *remberevens* from the first lecture. We take that definition and replace each use of $unit_{state}$ by $unit_{identity}$ and $star_{state}$ by $star_{identity}$. Then we get the following definition.

```
(define remberevens
  (λ (l)
    (cond
      ((null? l) (unit_identity '()))
      ((list?? (car l))
       ((star_identity (λ (a)
                         ((star_identity (λ (d) (unit_identity (cons a d))))
                          (remberevens (cdr l)))))
        (remberevens (car l))))
      ((odd? (car l))
       ((star_identity (λ (d) (unit_identity (cons (car l) d))))
        (remberevens (cdr l))))
      (else (remberevens (cdr l))))))

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

This is a pure solution because we have a very clean *unit* and *star*: both are obviously the identity function. It is trivial to revise the *identity* monad to use a pair whose *car* has the pure value.

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 $star_{state}$. We start our complete thirty-six step solution.

```
(define remberevensXcountevens
  (λ (l)
    (cond
      ((null? l) (unit_state '()))
      ((list?? (car l))
       ((star_state (λ (a)
                      ((star_state (λ (d) (unit_state (cons a d))))
                       (remberevensXcountevens (cdr l)))))
        (remberevensXcountevens (car l))))
      ((odd? (car l))
       ((star_state (λ (d) (unit_state (cons (car l) d))))
        (remberevensXcountevens (cdr l))))
      (else
       ((star_state (λ (__) (remberevensXcountevens (cdr l))))
        (λ (s) `(__ . ,(add1 s))))))))
```

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 *star*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 $[.../star_{state}]$. 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 $[.../star_{state}]$.

```
(define rememberevensXcountevens
  (λ (l)
    (cond
      ((null? l) ((λ (a) (λ (s) `(,a . ,s))) '()))
      ((list?? (car l))
       (((λ (sequel)
           (λ (ma)
             (λ (s)
               (let ((p (ma s)))
                 (let ((new-a (car p)) (new-s (cdr p)))
                   (let ((mb (sequel new-a)))
                     (mb new-s)))))))
         (λ (a)
           (((λ (sequel)
               (λ (ma)
                 (λ (s)
                   (let ((p (ma s)))
                     (let ((new-a (car p)) (new-s (cdr p)))
                       (let ((mb (sequel new-a)))
                         (mb new-s)))))))
             (λ (d) ((λ (a) (λ (s) `(,a . ,s))) (cons a d))))
           (rememberevensXcountevens (cdr l)))))
        (rememberevensXcountevens (car l))))
      ((odd? (car l))
       (((λ (sequel)
           (λ (ma)
             (λ (s)
               (let ((p (ma s)))
                 (let ((new-a (car p)) (new-s (cdr p)))
                   (let ((mb (sequel new-a)))
                     (mb new-s)))))))
         (λ (d) ((λ (a) (λ (s) `(,a . ,s))) (cons (car l) d))))
        (rememberevensXcountevens (cdr l))))
      (else
       (((λ (sequel)
           (λ (ma)
             (λ (s)
               (let ((p (ma s)))
                 (let ((new-a (car p)) (new-s (cdr p)))
                   (let ((mb (sequel new-a)))
                     (mb new-s)))))))
         (λ (a) (λ (s) `(,a . ,(add1 s)))))
        (rememberevensXcountevens (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*].

```
((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))))))
  (rememberevensXcountevens (cdr l))))
```

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

```
((odd? (car l))
 (λ (s)
   (let ((p ((rememberevensXcountevens (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*].

```
((odd? (car l))
 (λ (s)
   (let ((p ((rememberevensXcountevens (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*].

```
((odd? (car l))
 (λ (s)
   (let ((p ((rememberevensXcountevens (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)].

```
((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].

```
((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].

```
((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.

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

And so we begin.

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

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

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

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

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

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

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

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

24. [(*car p*)/*â*].

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

25. [.../*sequel̂*].

```
((list?? (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*].

```
((list?? (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. $[(cdr\ p)/\hat{s}]$.

```
((list?? (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. $[(car\ \hat{p})/new\text{-}\hat{a}]$, and $[(cdr\ \hat{p})/new\text{-}\hat{s}]$.

```
((list?? (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. $[.../m\hat{b}]$.

```
((list?? (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. $[(car\ \hat{p})/d]$.

```
((list?? (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. $[(cons\ (car\ p)\ (car\ \hat{p}))/a]$.

```
((list?? (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. $[(cdr\ \hat{p})/s]$.

```
((list?? (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)))
      ((list?? (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))))))
      ((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 remberevens×countevens
  (λ (l)
    (λ (s)
      (cond
        ((null? l) `(() . ,s))
        ((list?? (car l))
         (let ((p ((remberevens×countevens (car l)) s)))
           (let ((p̂ ((remberevens×countevens (cdr l)) (cdr p))))
             `(,(cons (car p) (car p̂)) . ,(cdr p̂)))))
        ((odd? (car l))
         (let ((p ((remberevens×countevens (cdr l)) s)))
           `(,(cons (car l) (car p)) . ,(cdr p))))
        (else
         (let ((p ((remberevens×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 *remberevens*×*countevens*.

36. Uncurry.

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

```
> (remberevens×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 $star_{state}$) 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 uses *bind* whereas I, like Moggi, use *star*[8]; Wadler shows how to extend an interpreter whereas I show how to extend "The Little Schemer" programs; Wadler assumes a reading knowledge of Haskell whereas I assume knowledge of functions as values and a reading knowledge of Scheme. In the final analysis, I believe my approach to be clearer for the novice and Wadler's appraoch to be clearer for the more sophisticated reader.

14 Acknowledgements

I have had conversations over the years with various people about monads, but some stand out as important as I developed my own way of explaining them. I want to thank, in alphabetical order, Michael Adams, David Bender, Will Byrd, Matthias Felleisen, Robby Findler, Steve Ganz, Ron Garcia, Roshan James, Ramana Kumar, Ed Kmett, Joe Near, Jiho Kim, Oleg Kiselyov, Anurag Mendhekar, Chung-Chieh Shan, Amr Sabry, Jeremy Seik, Jonathan Sobel, Larisse Voufo, and Mitch Wand. I am grateful for the tutorial papers by Phil Wadler, the tutorial by Jeff Newbern, and the stunningly clear paper by Eugenio Moggi that I mentioned above. I want to thank the c311/b521 dream team (Fall, 09) Lindsey Kuper, Nilesh Mahajan, Melanie Dybvig, and especially Adam Foltzer. This team convinced me that we should be able to teach this material to undergraduates.

[8]As pointed out above, this is basically a syntactic difference, but I chose *star* rather than *bind* because using *bind* looks too much like writing in continuation-passing style, and although the word "bind" works well once you understand the concept, it brings to mind too many associations like **let**, which can easily lull the reader into a false sense of understanding.