CSE 413, Fall 2012, Lazy Evaluation Summary

Notes adapted from CSE 341 notes by Dan Grossman, Fall 2011.

The primary focus of this lecture is powerful programming idioms related to *delaying evaluation* (using functions) and/or *remembering previous results* (using mutation). We will study *lazy evaluation*, *streams*, and *memoization*.

But first we need to introduce how mutable bindings work in Racket, the truth about cons cells (they are just pairs), and mutable cons cells (because the cons cells created by `cons` are not mutable).

Bindings are generally mutable: `set!` exists

While Racket encourages a functional-programming style with liberal use of closures and avoiding side effects, the truth is it has assignment statements. If `x` is in your environment, then `(set! x 13)` will *mutate* the binding so that `x` now maps to the value 13. Doing so affects all code that has this `x` in its environment. Pronounced "set-bang," the exclamation point is a convention to alert readers of your code that side effects are occurring that may affect other code. Here is an example:

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define z (f 4))
(define w c)
```

After evaluating this program, `z` is bound to 9 because the body of the function bound to `f` will, when evaluated, look up `b` and find `5`. However, `w` is bound to `7` because when we evaluated `(define c (+ b 4))`, we found `b` was `3` and, as usual, the result is to bind `c` to `7` regardless of how we got the `7`. So when we evaluate `(define w c)`, we get 7; it is irrelevant that `b` has changed.

You can also use `set!` for local variables and the same sort of reasoning applies: you have to think about *when* you look up a variable to determine what value you get. But programmers used to languages with assignment statements are all too used to that.

Mutating top-level bindings is particularly worrisome because we may not know all the code that is using the definition. For example, our function `f` above uses `b` and could act strangely, even fail mysteriously, if `b` is mutated to hold an unexpected value. If `f` needed to defend against this possibility it would need to avoid using `b` after `b` might change. There is a general technique in software development you should know: *If something might get mutated and you need the old value, make a copy before the mutation can occur.* In Racket, we could code this up easily enough:

```
(define f
   (let ([b b])
      (lambda (x) (* 1 (+ x b)))))
```

This code makes the `b` in the function body refer to a local `b` that is initialized to the global `b`.

(Note: You might not have seen this before, but scheme allows pairs of brackets `[` and `]` to be used in place of parentheses. The meaning is the same, but some people find this style more readable for things like definition pairs in `let` expressions.)

But is this as defensive as we need to be? Since `*` and `+` are just variables bound to functions, we might want to defend against them being mutated later as well:

```
(define f
   (let ([b b]
         [+ +]
         [* *])
      (lambda (x) (* 1 (+ x b)))))
```

Matters would get worse if `f` used other helper functions: Making local copies of variables bound to the functions would not be enough unless those functions made copies of all their helper functions as well.

The previous discussion is *not* something that will affect most of your Racket programming (since you won't be doing things like that, right?), but it is useful to understand what `set!` means and how to defend against mutation by making a copy. The point is that the possibility of mutation, which Racket often avoids, makes it very difficult to write correct code.

The truth about cons cells

So far, we have used `cons`, `null`, `car`, `cdr`, and `null?` to create and access lists. For example, `(cons 14 (cons #t null))` makes the list `'(14 #t)` where the quote-character shows this is printing a list value, not indicating an (erroneous) function call to 14.

But the truth is *cons cells are just pairs* where you get the first part with `car` and the second part with `cdr`. So we can write `(cons (+ 7 7) #t)` to produce the pair `'(14 . #t)` where the period shows that this is *not* a list. A list is, by convention and according to the `list?` predefined function, is either `null` or a pair where the cdr (i.e., second component) is a list. A cons cell that is not a list is often called an *improper list*, especially if it has nested cons cells in the second position, e.g., `(cons 1 (cons 2 (cons 3 4)))` where the result prints as `'(1 2 3 . 4)`. Most list functions like `length` will give a run-time error if passed an improper list.

What are improper lists good for? The real point is that pairs are a generally useful way to build data with multiple pieces. And in a dynamically typed language, all you need for lists are pairs and some way to recognize the end of the list, which in Racket is done with the `null` constant (which prints as `'()`).

Mutable cons cells

We now revisit mutation. Consider this code:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
```

The `set!` changes `x` to refer to a new different cons cell. Since `y` refers to the old cons cell (we looked up `x` prior to the mutation to initialize `y`), we still have `y` evaluating to `'(14 null)`. What if we want to mutate the *contents of a cons cell* so that any variables or data structures referring to the cons cell see the new value? This would be analogous to assigning to an object field in Java (`x.cdr = 42`), which is entirely different from an assigning to a variable that refers to an object (`x = new Cons(42,null)`).

In Racket, cons cells are immutable, so this is not possible. (In Scheme, it is possible using primitives `set-car!` and `set-cdr!`.) Hence we can continue to enjoy the benefits of knowing that cons cells cannot be mutated by other code in our program. It has another somewhat subtle advantage: The Racket implementation can be clever enough to make `list?` a constant-time operation since it can store with every cons cell whether or not it is a proper list when the cons cell is created. This cannot work if cons cells are mutable because a mutation far down the list could turn it into an improper list.

If we want mutable pairs, though, Racket is happy to oblige with a different set of primitives:

- `mcons` makes a mutable pair

- `mcar` returns the first component of a mutable pair
- `mcdr` returns the second component of a mutable pair
- `mpair?` returns `#t` if given a mutable pair
- `set-mcar!` takes a mutable pair and an expression and changes the first component to be the result of the expression
- `set-mcdr!` takes a mutable pair and an expression and changes the second component to be the result of the expression

Since some of the powerful idioms in the rest of the lecture use mutation to store previously computed results, we will find mutable pairs useful.

Delayed Evaluation

A key semantic issue for a language construct is *when are its subexpressions evaluated*. For example, in Racket (and similarly in Java and most conventional languages), given `(e1 e2 ... en)` we evaluate the function arguments `e2`, ..., `en` once before we execute the function body and given a function `(lambda (...) ...)` we do not evaluate the body until it is called. We can contrast this rule ("evaluate arguments in advance") with how `(if e1 e2 e3)` works: we do *not* evaluate both `e2` and `e3`. This is why:

```
(define (bad-if x y z) (if x y z))
```

is a function that *cannot* be used wherever you use an if-expression; the rules for evaluating subexpressions are fundamentally different. For example, this function would never terminate since every call makes a recursive call:

```
(define (factorial-wrong x)
  (bad-if (= x 0)
          1
          (* x (factorial-wrong (- x 1)))))
```

However, we can use the fact that function bodies are not evaluated until the function gets called to make a more useful version of an "if function":

```
(define (good-if x y z) (if x (y) (z)))
```

Now wherever we would write `(if e1 e2 e3)` we could instead write `(good-if e1 (lambda () e2) (lambda () e3))`. The body of `good-if` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`. So this function is correct (for non-negative arguments):

```
(define (factorial x)
  (good-if (= x 0)
          (lambda () 1)
          (lambda () (* x (factorial (- x 1))))))
```

Though there is certainly no reason to wrap Racket's "if" in this way, the general idiom of using a zero-argument function to *delay evaluation* (do not evaluate the expression now, do it later when/if the zero-argument function is called) is very powerful. As convenient terminology/jargon, when we use a zero-argument function to delay evaluation we call the function a *thunk*. You can even say, "thunk the argument" to mean "use `(lambda () e)` instead of `e`".

The rest of lecture considers three programming idioms; thunks are crucial in two of them and the third is related.

Lazy Evaluation

Suppose we have a very large computation that we know how to perform but we do not know if we need to perform it. The rest of the application knows where it needs this computation and there may be a few different places. If we thunk, then we may repeat the large computation many times. But if we do not thunk, then we will perform the large computation even if we do not need to. To get the "best of both worlds," we can use a programming idiom known by a few different (and perhaps technically slightly different) names: lazy-evaluation, call-by-need, promises. The idea is to use mutation to remember the result from the first time we use the thunk so that we do not need to use the thunk again.

One simple implementation in Racket would be:

```
(define (delay-eval f)
  (mcons #f f))

(define (force-eval th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

We can create a thunk `f` and pass it to `delay-eval`. This returns a pair where the first field indicates we have not used the thunk yet. Then `force-eval`, if it sees the thunk has not been used yet, uses it and then uses mutation to change the pair to hold the result of using the thunk. That way, any future calls to `force-eval` with the same pair will not repeat the computation. Ironically, while we are using mutation in our *implementation*, this idiom is quite error-prone unless the thunk passed to `delay-eval` does not have side effects, since those effects will occur at most once and it may be difficult to determine when the first call to `force-eval` will occur.

Consider this silly example where we want to multiply the result of two expressions `e1` and `e2` using a recursive algorithm (of course you would really just use `*` and this algorithm does not work if `e1` produces a negative number):

```
(define (mult x y)
 (cond [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (mult (- x 1) y))]))
```

Now calling `(mult e1 e2)` evaluates `e1` and `e2` once each and then does 0 or more additions. But what if `e1` evaluates to 0 and `e2` takes a long to compute? Then evaluating `e2` was wasteful. So we could thunk it:

```
(define (mult x y-thunk)
 (cond [(= x 0) 0]
       [(= x 1) (y-thunk)]
       [#t (+ y (mult (- x 1) (y-thunk)))]))
```

Now we would call `(mult e1 (lambda () e2))`. This works great if `e1` evaluates 0, fine if `e1` evaluates to 1, and terribly if `e1` evaluates to a large number. After all, now we evaluate `e2` on every recursive call. So let's use `delay-eval` and `force-eval` to get the best of both worlds:

```
(mult e1 (let ([x (delay-eval (lambda () e2))]) (lambda () (force-eval x))))
```

Notice we create the delayed computation once before calling `mult`, then the first time the thunk passed to `mult` is called, `force-eval` will evaluate `e2` and remember the result for future calls to `force-eval x`. An alternate approach that might look simpler is to rewrite `mult` to expect a result from `delay-eval` rather than an arbitrary thunk:

```
(define (mult x y-promise)
 (cond [(= x 0) 0]
       [(= x 1) (force-eval y-promise)]
       [#t (+ (force-eval y-promise) (mult (- x 1) y-promise))]))

(mult e1 (delay-eval (lambda () e2)))
```

Some languages, most notably Haskell, use this approach for all function calls, i.e., the semantics for function calls is different in these languages: If an argument is never used it is never evaluated, else it is evaluated only once. This is called *call-by-need* whereas all the languages we will use are *call-by-value* (arguments are fully evaluated before the call is made).

Streams

A stream is an infinite sequence of values. We obviously cannot create such a sequence explicitly (it would literally take forever), but we can create code that knows how to produce the infinite sequence and other code that knows how to ask for however much of the sequence it needs.

Streams are very common in computer science. You can view the sequence of bits produced by a synchronous circuit as a stream, one value for each clock cycle. The circuit does not know how long it should run, but it can produce new values forever. The UNIX pipe (`cmd1 | cmd2`) is a stream; it causes `cmd1` to produce only as much output as `cmd2` needs for input. Web programs that react to things users click on web pages can treat the user's activities as a stream — not knowing when the next will arrive or how many there are, but ready to respond appropriately. More generally, streams can be a convenient division of labor: one part of the software knows how to produce successive values in the infinite sequence but does not know how many will be needed and/or what to do with them. Another part can determine how many are needed but does not know how to generate them.

There are many ways to code up streams; we will take the simple approach of representing a stream as a thunk that when called produces a pair of (1) the first element in the sequence and (2) a thunk that represents the stream for the second-through-infinity elements. Defining such thunks typically uses recursion. Here are three examples:

```
(define ones (lambda () (cons 1 ones)))
(define nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(define powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Given this encoding of streams and a stream `s`, we would get the first element via `(car (s))`, the second element via `(car ((cdr (s))))`, the third element via `(car ((cdr ((cdr (s))))))`, etc. Remember parentheses matter: `(e)` calls the thunk `e`.

We could write a higher-order function that takes a stream and a predicate-function and returns how many stream elements are produced before the predicate-function returns true:

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                  (if (tester (car pr))
                      ans
                      (f (cdr pr) (+ ans 1)))))])
    (f stream 1)))
```

As an example, `(number-until powers2 (lambda (x) (= x 16)))` evaluates to 4.

As a side-note, all the streams above can produce their next element given at most their previous element. So we could use a higher-order function to abstract out the common aspects of these functions, which lets us put the stream-creation logic in one place and the details for the particular streams in another. This is just another example of using higher-order functions to reuse common functionality:

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg)))))])
    (lambda () (f arg))))
(define ones (stream-maker (lambda (x y) 1) 1))
(define nats  (stream-maker + 1))
(define powers-of-two (stream-maker * 2))
```

Memoization

An idiom related to lazy evaluation that does not actually use thunks is *memoization.* If a function does not have side-effects, then if we call it multiple times with the same argument(s), we do not actually have to do the call more than once. Instead, we can look up what the answer was the first time we called the function with the argument(s).

Whether this is a good idea or not depends on trade-offs. Keeping old answers in a table takes space and table lookups do take some time, but compared to reperforming expensive computations, it can be a big win. Again, for this technique to even be *correct* requires that given the same arguments a function will always return the same result and have no side-effects. So being able to use this *memo table* (i.e., do memoization) is yet another advantage of avoiding mutation.

To implement memoization we do use mutation: Whenever the function is called with an argument we have not seen before, we compute the answer and then add the result to the table (via mutation).

As an example, let's consider 3 versions of a function that takes an `x` and returns fibonacci(x). A fibonacci number is a well-known definition that is useful in modeling populations and such.) A simple recursive definition is:

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `(fibonaccci 30)`, and `(fibonacci 40)` takes a thousand times longer than that, and `(fibonacci 10000)` would take more seconds than there are particles in the universe. Now, we could fix this by taking a "count up" approach that remembers previous answers:

```
(define (fibonacci x)
  (letrec ([f (lambda (acc1 acc2 y)
                (if (= y x)
                    (+ acc1 acc2)
                    (f (+ acc1 acc2) acc1 (+ y 1))))])
    (if (or (= x 1) (= x 2))
        1
        (f 1 1 3))))
```

This takes linear time, so `(fibonacci 10000)` returns almost immediately (and with a very large number), but it required a quite different approach to the problem. With memoization we can turn `fibonacci` into an efficient algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you often learn about in advanced algorithms courses. Here is the version that does this memoization:

```
(define fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

It is essential that different calls to `f` use the *same* mutable memo-table: if we create the table inside the call to `f`, then each call will use a new empty table, which is pointless. But we do not put the table at top-level just because that would be bad style since its existence should be known only to the implementation of `fibonacci`.

For a large table, using a list and Racket's `assoc` function may be a poor choice, but it is fine for demonstrating the concept of memoization.

Why does this technique work to make `(fibonacci 10000)` complete quickly? Because when we evaluate `(f (- x 2))` on any recursive calls, the result is already in the table, so there is no longer an exponential number of recursive calls. This is much more important than the fact that calling `(fibonacci 10000)` a second time will complete even more quickly since the answer will be in the memo-table.