CSE 341, Spring 2008, Lecture 14 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

Scheme top-level (again):

In ML, a top-level `fun` binding's environment included only itself and earlier bindings. This leads to a straightforward semantics, but it can be inconvenient (having to organize your functions in a certain order) or limiting (which is why we had different syntax for mutual recursion). In Scheme, earlier functions can refer to later bindings, but this comes at the cost of more complicated rules, especially since Scheme allows any binding to be *mutated* (with `set!`). You may assume what essentially all Scheme programmers do — that top-level bindings are not mutated. Nonetheless, it is worth investigating how mutation of top-level definitions affects program correctness because it motivates an important programming idiom that comes up throughout computer science (particularly in operating systems and security): *If data/code might change and you need to assume it doesn't, make a local copy.*

This code fails because when we evaluate the `y` on the first line there is no `x` in the environment:

```
(define x y)
(define y 1)
```

However, this code defines `f1` as a zero-argument function that returns 1 when called. That is because all top-level definitions are included in a function's environment, even top-level definitions added after the function is defined.

```
(define f1 (lambda () y))
(define y 1)
```

However, we cannot really say `f1` always returns 1, since if there is another top-level definition of `y`, it will have the effect of mutation.

```
(define y 7) ; now (f1) evaluates to 7
```

Similarly, any code can use `set!` to change what value anything in the environment has.

Consider a more realistic example with a simple curried exponentiation function and a cube function that uses partial application:

```
(define (pow y)
  (lambda (x)
    (if (= y 0)
        1
        (* x ((pow (- y 1)) x)))))

(define cube (pow 3))
```

For the corresponding ML code, we know any call to `cube` will "do the right thing", but in Scheme that assumes that nothing is mutated/redefined to break `pow`. If we later do `(define pow 6)` then the recursive call in `pow`'s body will not be recursive at all — it will fail because `pow` is no longer bound to a function. If we later do `(define (pow y) (lambda (x) 2))` we will produce the wrong answer. And if we later do `(define - +)` (after all, `-` is just a binding so we can redefine it to be addition), then `pow` will go into an infinite loop.

Though you should not bother in Scheme, the general solution to avoiding code breaking due to mutation that is beyond your control is to make "private" copies of what must not change. For example, we could define a version of `pow` in which there was a local environment that bound all the helper functions we needed (including `=`, `*`, and `-`). That will work because top-level redefinitions will not change the local environment we created.

Delayed Evaluation:

A key semantic issue for a language construct is *when are its subexpressions evaluated*. For example, in Scheme (and similarly in Java and ML), 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 (my-if-bad 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.

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 (my-if x y z) (if x (y) (z)))
```

Now wherever we would write `(if e1 e2 e3)` we could instead write `(my-if e1 (lambda () e2) (lambda () e3))`. The body of `my-if` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`.

Though there is certainly no reason to wrap Scheme's "if" in this way, the general idiom of using a zero-argument function to *delay evaluation* (do not evaluate `e2` 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 3 programming idioms; thunks are crucial in two of them and the third is similar.

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 (really) to remember the result from first time we use the thunk so that we do not need to use the thunk again.

One simple implementation in Scheme would be:

```
(define (my-delay f)
  (cons #f f))

(define (my-force th)
  (if (car th)
      (cdr th)
      (begin (set-car! th #t)
             (set-cdr! th ((cdr th)))
             (cdr th))))
```

We can create a thunk `f` and pass it to `my-delay`. This returns a pair where the first field indicates we have not used the thunk yet. Then `my-force`, 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 `my-force` with the same pair will not repeate the computation.

Consider this silly example where 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 (my-mult x y)
 (cond [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (my-mult (- x 1) y))]))
```

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

Now we would call (my-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 my-delay and my-force to get the best of both worlds. We can just call:
(my-mult e1 (let ([x (my-delay (lambda () e2))]) (lambda () (my-force x)))) Notice we create the delayed computation once before calling my-mult, then the first time the thunk passed to my-mult is called, my-force will evaluate e2 and remember the result for future calls to my-force x. An alternate approach that might look simpler is to rewrite my-mult to expect a result from my-delay rather than an arbitrary thunk:

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

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

Some languages (most notably Haskell and Miranda) 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.

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.

That said, 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 mathematic formula that is useful in modeling populations and such.) A simple recursive definition is:

```
(define (fibonacci1 x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci1 (- x 1))
         (fibonacci1 (- x 2)))))
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `(fibonaccci1 30)`, and `(fibonacci1 40)` takes a thousand times longer than that, and `(fiboannica1 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 (fibonacci2 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 `(fibonacci2 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 `fibonacci1` into a linear algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you learn about in CSE421. Here is the version that does this memoization:

```
(define fibonacci3
  (letrec([memo (list (cons 1 1) (cons 2 1))]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (let ([new-ans (+ (f (- x 1))
                                       (f (- x 2)))])
                       (begin
                         (set! memo (cons
                                     (cons x new-ans)
                                     memo))
                         new-ans)))))])
    f))
```