MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 23 — 5/10/2006
Lazy Evaluation

Applicative vs Normal Order evaluation

In applicative order execution (like regular Scheme), all procedure arguments are evaluated before applying the procedure. In normal order execution, procedure arguments are evaluated after applying the procedure, and then only if the result is needed to complete the evaluation of the procedure.

If there are no side effects or mutation, (another way of saying this would be to say that if all expressions were idempotent, meaning that you could evaluate the same expressions repeatedly without and other effects or different results), then the final returned value will be the same for either normal or applicative order application.

However, with mutation, the results will not be the same:

```
(define a 1)
(define b 1)

(define (foo x y)
  (+ x y y))

(define res (foo
              (begin (set! a (+ a 1))
                     a)
              (begin (set! b (* b 2))
                     b)))

(define v (cons a (cons b '())))
(cons res v)
```

In an applicative order Scheme, applying `(cons res v)` will return `(6 2 2)`. What will a normal order evaluation return? (no memoization yet). Draw diagrams for both.

A normal order (lazy) evaluator would make it easy to define procedures that would need to be special forms in standard Scheme:

```
(define (unless test a b)
   (if test b a))
```

In an applicative order evaluator `unless` would need to be a special form, because we don't want to evaluate both `a` and `b`.

Lazy Evaluator

Let's change meval to be fully lazy. (Lecture notes covered a hybrid example – we won't use that here). Only a few changes are needed:

1. In m-apply, add an environment argument, and if the procedure is a primitive, force the argument values and apply the primitive procedure.
2. Also in m-apply, if the procedure is a compound, delay evaluating the arguments – extend the environment of the procedure with *thunks* – promises to evalue the arguments later in the environment passed to apply.
3. In m-eval, if the procedure is an application, force the value of the procedure (so that mapply can tell whether it is a primitive or compound), but pass the expressions for the arguments to apply.
4. helper procedures for dealing with thunks. `actual-value` and `force-it`. Also, the driver loop needs to use `actual-value` to show the final answer.

Memoization

In the top example, the expression for `y` was evaluated twice, which is both wasteful, and changes the result of the expressions. Instead of evaluating a thunk each time the value is needed, we can force the value only once, and thereafter remember the value.

Redo the beginning example in a normal order evaluator with memoization. Does the result look like an applicative order or a normal order without memoization? What set of expressions would look different?

Streams Intro

Consider evaluating the following in a lazy (normal order) Scheme:

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
(define (cdr z)
  (z (lambda (p q) q)))

(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))

(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))

(define foo (cons 1 foo))

(define bar (cons 1 (add-lists foo bar)))
```

1. What would `(list-ref foo 2)` return? What about `(list-ref foo 12345)`?.
2. Now look at `bar`. `(list-ref bar 2)`? `(list-ref bar 20)`?
3. Define some other interesting value using similar ideas as above (We'll see more on friday, including ways to do this without a fully lazy evaluator)