MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6001—Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 8: Solutions

Exercise 2. Demonstrate how, using lazy parameters, you can define three argument *or* as a procedure.

The solution we had in mind is exactly the code you might have written at the start of the course before you knew the difference between a procedure and a special form, albeit with some parameters marked as lazy:

```
(define (or a (b lazy) (c lazy))
  (if a
      a
      (if b
          b
          (if c
              c
              #f))))
```

Note that the first parameter is always evaluated, so nothing is gained by making it lazy. Here are some applications that show that the procedure correctly avoids evaluating arguments whose values are not needed:

```
MEVAL=> (or 1 2 3)
;;M-value:
1

MEVAL=> (or #f 2 3)
;;M-value:
2

MEVAL=> (or 1 (1 1) 3)
;;M-value:
1
```

This solution is not quite right, however: an argument may be evaluated twice:

```
MEVAL=> (define x 0)
;;M-value:
#[unspecified-return-value]

MEVAL=> (define (inc) (begin (set! x (+ x 1)) x))
;;M-value:
#[unspecified-return-value]

MEVAL=> (or #f (inc) 3)
;;M-value:
2
```

To ensure that each argument is evaluated at most once, we can declare the lazy parameters as lazy-memo *so that the argument values are memoized:*

```
(define (or a (b lazy-memo) (c lazy-memo))
  (if a
      a
      (if b
          b
          (if c
              c
              #f))))
```

Now the double evaluation problem goes away:

```
MEVAL=> (define x 0)
;;M-value:
#[unspecified-return-value]

MEVAL=> (define (inc) (begin (set! x (+ x 1)) x))
;;M-value:
#[unspecified-return-value]

MEVAL=> (or #f (inc) 3)
;;M-value:
1
```

Another way to solve this problem is to evaluate the argument and bind it to a local variable with a let. *Since* let *desugars to a lambda with lazy parameters, we have to find some way to force evaluation. Here is one solution; it exploits the fact that primitive procedures such as* cons *are strict:*

```
 (define (or a (b lazy) (c lazy))
   (if a
       a
       (let ((vb (force b)))
          (if vb
              vb
              (let ((vc (force c)))
                 (if vc
                     vc
                      #f))))
(define (force x ) (car (cons x nil))
```

Exercise 3. Define an or->desugared procedure which transforms an arbitrary or expression, which may have a varying number of arguments as in real Scheme, into an equivalent expression that meval can handle. Using this procedure, extend meval so or is implemented as a derived expression, ie, as "syntactic sugar".

Our solution is modeled on the treatment of let *expressions. We add a case to the condition in* meval *that first desugars the expression then evaluates it:*

```
((or? exp) (meval (or->desugared exp) env))
```

An or expression of the form

```
(or e1 e2 ...)
```

is desugared into a nested if statement with lambdas *to bind the computed values:*

```
((lambda (v) (if v v (lambda (v) (if v v (... en) )) e2 ) )) e1)
```

The desugaring procedure constructs this:

```
(define (or->desugared exp)
  (define (helper exp)
    (if (null? exp)
        #f
        (list
             (make-lambda '(v)
                  (make-if 'v 'v (helper (cdr exp))))
                  (car exp))))
  (helper (cdr exp)))
```

Here are the auxililary procedures for constructing lambda expressions and recognizing an or-expression:

```
(define (make-lambda param-list body)
        (list 'lambda param-list body))

(define (or? exp) (tagged-list? exp 'or))
```

Our solution has a subtle bug. If the variable v *appears inside one of the expressions* e2, e3, ... *, the new binding will incorrectly shadow the old binding. This phenomenon is called "variable capture". One way to solve it is to choose a variable name that is not used in any of the expressions. A more elegant solution is to form thunks for each of the expressions* e2, e3, *…, in the outermost environment, and then force the thunk when the value is required.*

Exercise 4. Give an example of what would go wrong if the evaluator's variable pair? were bound to Scheme's pair? rather than the special procedure mpair?. Define a procedure mproc? which tells whether a value is the evaluator's representation of a procedure. Add a procedure? predicate to the evaluator.

A compound procedure, which the evaluator represents internally as a list consisting or parameters, body and environment, would be recognized as a cons cell:

```
MEVAL=> (pair? (lambda (x) x))
;;M-value:
#t
```

But with pair? *bound instead to* mpair?*, procedures are correctly distinguished from data structures:*

```
MEVAL=> (pair? (lambda (x) x))
;;M-value:
#f
```

Here is the procedure that determines whether a value represents a procedure.

```
(define (mproc? v) (or (primitive-procedure? v)
                       (compound-procedure? v)))
```

Having bound procedure? *to* mproc? *in the initial environment as instructed:*

```
(define (initial-environment)
    (extend-environment
      '(pair?
        procedure?
        ...
       (list
          mpair?
          mproc?
          ... )
```

we can now try to use it:

```
MEVAL=> (procedure? 3)
;;M-value:
#f

MEVAL=> (procedure? cons)
;;M-value:
#t

MEVAL=> (procedure? (lambda (x) x))
;;M-value:
#t
```

Exercise 5A. One proposal for implementing lazy lists uses a procedure lazy-cons rather than cons:

```
(define (lazy-cons (a lazy) (b lazy)) (cons a b))
```

Explain why this does not work.

It fails because the arguments to lazy-cons *are always evaluated in the call to the primitive* cons.

Exercise 5B. Modify the evaluator so that

```
(cons <expr1> <expr2>)
```

returns a cons-cell whose car and cdr are the unforced values of <expr1> and <expr2>.

As suggested, there are two ways to do this. The better way is to treat cons as a special kind of built-in procedure; applying the procedure can be given the special behaviour, but the procedure can be passed as value like any other.

We create a special value to represent the built-in cons *procedure:*

```
(define *built-in-cons* (list 'built-in-cons))
```

and replace the cons procedure in the initial environment with this value. Now we modify m-apply*, so that it treats application of the built-in* cons *procedure differently:*

```
(define (m-apply procedure aprocs env)
   (cond ((eq? procedure *built-in-cons*)
               (cons (delay-it (car aprocs) env) (delay-it (cadr aprocs) env))))
         ((primitive-procedure? procedure)
          ...
```

Alternatively, to treat cons *as a special form, we remove the binding of* cons *from the initial global environment, add a new case to* meval*:*

```
((cons? exp)
     (cons (delay-it(lambda (e) (meval (cadr exp) e)) env)
           (delay-it(lambda (e) (meval (caddr exp) e)) env)))
```

and define a procedure for recognizing a cons *expression:*

```
(define (cons? exp) (tagged-list? exp 'cons))
```

Exercise 5C. Try out your lazy lists on the prime stream of p.330 of SICP.

Here are the appropriate procedure definitions:

```
(define (integers-starting-from n)
 (cons n (integers-starting-from (+ n 1))))

(define (divisible? x y) (= (remainder x y) 0))

(define (filter proc l)
  (cond ((null? l) '())
    ((proc (car l))(cons (car l) (filter proc (cdr l))))
    (else (filter proc (cdr l)))))

(define primes (cons 2 (filter prime? (integers-starting-from 3))))

(define (square x) (* x x))

(define (prime? n)
   (define (iter ps)
     (cond ((> (square (car ps)) n) #t)
       ((divisible? n (car ps)) #f)
       (else (iter (cdr ps)))))
   (iter primes))
```

and here is a sample run:

```
MEVAL=> (car primes)
;;M-value:
2

MEVAL=> (car (cdr primes))
;;M-value:
3

MEVAL=> (car (cdr (cdr primes)))
;;M-value:
5

MEVAL=> (car (cdr (cdr (cdr primes))))
;;M-value:
7

MEVAL=> (car (cdr (cdr (cdr (cdr primes)))))
;;M-value:
11
```

Exercise 6. Extend the evaluator so it supports define-class (from problem set 7) as a special form.

This is trivial. The new case simply applies the transformation procedure (given in the code of PS7), and then evaluates the resulting expression:

```
((define-class? exp) (meval (transform-class-definition (class-definition exp)) env))
```

The two auxiliary procedures used are:

```
(define (define-class? exp) (tagged-list? exp 'define-class))
(define (class-definition exp) (cdr exp)
```

The class definitions are exactly as before, except that there is no need for the quotation surrounding the body of the definition. We can write, for example:

```
(define-class (class-name account)
              (parts ())
              (slots ((balance 0)))
              (constructor ...)
              (methods ...))
```

(In fact, the code from Problem Set 7 cannot unfortunately be evaluated directly using our interpreter, since it makes use of dotted pair notation for varying length argument lists which our interpreter does not support.)

Exercise 7A. Add the for-list construct as a special form.

We add a new case to the conditional:

```
((for-list? exp) (eval-for-list exp env))
```

The auxiliary procedures are:

```
(define (for-list? exp)(eq? (car exp) 'for-list))

(define (eval-for-list exp env)
  (define var (cadr exp))
  (define listarg (meval (caddr exp) env))
  (define body (cadddr exp))
  (define (iter l)
    (if (null? l)
    the-unspecified-value
    (begin
        (define-variable! var (car l) env)
        (meval body env)
        (iter (force-it (cdr l))))))
  (iter listarg))
```

The code of the eval-for-list *procedure mirrors exactly the evaluation rule for the* for-list. *First the list argument is evaluated. Then the body is repeatedly evaluated with the variable bound to each element of the list in succession.*

Note the explicit forcing of the rest of the list on the recursive call to iter. *Without this, evaluation will fail if the rest of the list is a delayed object; including it ensures that evaluation really is equivalent to the desugared form, so that it works even for a lazy list:*

Here's a sample run:

```
MEVAL=>(for-list a '(1 2 3) (display (* 2 a)))246
;;M-value:
(the-unspecified-value)

MEVAL=> (define (take n l) (if (eq? 0 n) '() (cons (car l) (take (- n 1) (cdr l)))))
;;M-value:
(the-unspecified-value)


MEVAL=> (for-list a (take 5 primes) (display a))235711
;;M-value:
(the-unspecified-value)
```