MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Friday, February 19

1. Abstraction using Higher Order Procedures

Let's take a look at using abstraction on common patterns.

`(* 2 5)`	`(double 5)`	`(* 3 2)`	`(triple 2)`
`(* 2 8)`	`(double 8)`	`(* 3 17)`	`(triple 17)`
`(* 2 54)`	`(double 54)`	`(* 3 4.1)`	`(triple 4.1)`

`(define double`
`  (lambda (x)`
`    (* 2 x)))`
	`(define double`
	`  (make-mult 2))`
		`(define triple`
		`  (lambda (x)`
		`    (* 3 x)))`
			`(define triple`
			`  (make-mult 3))`

```
(define make-mult
  (lambda (n)
    (lambda (x)
      (* n x))))
```

3. Higher Order Procedures

Write a function `swap` that takes a function `f`, and returns a function, that takes two arguments, and returns `f` with the variables swapped: `(f x y) == ((swap f) y x)` For example, `((swap -) 4 5)` ⇒ `1`.

```
(define swap
  (lambda (f)
    (lambda (x y)
      (f y x)))
)
```

2. Composing Procedures

Now try to write the function `compose` that takes two functions, `f` and `g`, and returns a function, that takes one argument, and composes `f` and `g` on that argument.

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))
)
```

Let's trace through the evaluation of the following expression:

```
((compose double cube) 3)
((λ (f g) (λ (x) (f (g x)))) double cube) 3)
((λ (x) (double (cube x))) 3)
(double (cube 3))
54
```

Notice that there's no magic here. We just used the same rules for evaluation that we've been using all along – the substitution model!

Using `compose`, define the function `^5/2` which takes a number `x` and computes $x^{5/2}$.

```
(define ^5/2  (compose sqrt cube)                          )
```

5. Repeated Composition of Procedures

We saw how to compose two procedures to produce another procedure. For example, we can define the following.

```
(define fourth-power (compose square square))
(define eight-power (compose square (compose square square)))
... and so on ...
```

Let's write a (very strange) function called `repeated` that takes a function `f` and an integer `n`, and composes `f`, `n` times. For example:

```
(define fourth-power (repeated square 2))
(define eight-power (repeated square 3))
... and so on ...

(define (repeated proc n)
  (if (= n 0)
      (lambda (x) x)
      (compose proc (repeated proc (- n 1))))
)
```

Let's look at a simple example:

```
(define fourth-power (repeated square 2))
(repeated square 2)
(compose square  (repeated square 1))
(compose square  (compose square  (repeated square 0)))
(compose square  (compose square  (lambda (x) x)))
...
```

6. Iterative Repeated

Guess what.. Now that we've written the recursive version of `repeated`, let's write the iterative version.

```
(define (repeated proc n)
  (define (iter n ans)
     (if (= n 0)
         ans
         (iter (- n 1) (compose f ans))))
  (iter n (lambda (x) x))
)
```

7. More Higher-Order Procedures

Write a function `snoc` that takes two arguments `a` and `b` and returns a function, that when called with `#t` returns `a` and when called with `#f` returns `b`.

```
(define snoc
  (lambda (a b)
    (lambda (x)
       (if x a b)))
)
```

What do we have once we define the following?

```
(define (rac x) (x #t))
(define (rdc x) (x #f))
```

Here's an even more elegant (albeit more obscure) way of doing the same thing. Can you figure out how this is working?

```
(define snoc (lambda (x y) (lambda (f) (f x y))))
(define rac (lambda (p) (p (lambda (a b) a))))
(define rdc (lambda (p) (p (lambda (a b) b))))
```