Quiz

Wednesday 7:30 to 9:30 in 4-[2/3]70
One sheet of notes double-sided
Quiz review… Wednesday recitation
Conflict exam on Friday:
Let Donna know right now!
dkauf@mit.edu
Because she will be away most of next week
LA quiz review: Sun Oct 2 at 7pm in 4-149
Mon Oct 3 in 4-159 at 7pm
Old quiz subjects are on the web

DrScheme upgrade

Get version 299.400 which, I believe, fixes the save as .scm issue.

In general be careful, verify with another text editor before closing DrScheme.

(compose f g)

Returns the function f(g(x))
(assume f and g take one variable)

```
((compose sqrt square) -4) → 4
(define (compose f g)
  (lambda(x) (f (g x))))
```

Type?

$(B \to C), (A \to B) \to (A \to C)$

```
((compose square square) 2)
16
```

Analyze by substitution

```
((lambda(x) (square (square x))) 2)
(square (square 2))
(square 4)
16
```

Repeated composition: (repeated f n)

Returns the function $x \to f(f(f\ldots(x)))$

```
((repeated square 4) 2) → 65536
```

Use compose

```
(define (identity x) x)
(define (repeated f n)
  (if (= n 0) identity
    (compose f (repeated f (- n 1)))))
```

Repeated composition: (repeated f n)

```
(define (repeated f n)
  (if (= n 0) identity
    (compose f (repeated f (- n 1)))))
```

Analyze by substitution (be lazy, use 3)

```
((repeated square 3) 2)
((if (= 3 0) identity (compose square (repeated square (- 3 1)))) 2)
((compose square (repeated square 2)) 2)
((compose square (if (= 2 0) identity (compose square (repeated square ( - 2
  1)))) 2)
((compose square (compose square (repeated square 1))) 2)
((compose square (compose square (if (= 1 0) identity (compose square (repeated
  square (- 1 1))))) 2)
((compose square (compose square (compose square (repeated square 0)))) 2)
((compose square (compose square (compose square (if (= 0 0) identity (compose
  square (repeated square (- 0 1)))))))) 2)
((compose square (compose square (compose square identity)))))) 2)
((compose square (compose square (lambda(x)(square (identity x)))))) 2)
((compose square (lambda(x)(square ((lambda(x)(square (identity x))) x)))) 2)
((lambda(x)(square (((lambda(x)(square ((lambda(x)(square (identity x))) x))))
  x)) 2)
(square (((lambda(x)(square ((lambda(x)(square (identity x))) x)))) 2))
(square (square ((lambda(x)(square (identity x))) 2)))
(square (square (square (identity 2))))
…
65536
;(formally, identity and square should be replaced by the actual procedure…)
```

Iterative version!

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

Generalized version

Note that repeated composition is similar to the definition of multiplication with multiple additions and the definition of exponentiation with multiple multiplications. Define a higher-order generalized-repeated such that you can do:

```
(define mult (generalized-repeated + 0))
(define exp (generalized-repeated * 1))
(define repeated (generalized-repeated
  compose identity))
```

And here we go:

```
(define (generalized-repeated op neutral)
  (define helper (lambda(x n)
    (if (= n 0) neutral
      (op x (helper x (- n 1))))))
```

Swap: takes procedure of two arguments, returns procedure where arguments are swapped

```
((swap /) 3 6) → 2
(define swap
  (lambda(f)(lambda (x y) (f y x))))
```

Type?

(A B→ C) → (B, A) → C

Define (specialize f arg)

takes a function of two variable and returns a function of one variables by always using arg as the second variable

```
(define half (specialize / 2))
(half 6) → 3
(define (specialize f arg)
  (lambda(x) (f x arg))
```

Type of specialize?

(A, B → C), B → (A → C)

Using specialize

Define positive? that checks if a number is greater than 0

```
(define positive? (specialize > 0))
```

Define inverse by specializing /

```
(define inverse
  (specialize (swap /) 1))
```

Careful, we want to specialize the first argument, and specialize only knows how to specialize the second one. Hence the use of swap

Now define (specialize-to-0 proc)

It takes a procedure of two arguments and returns a procedure of one argument where proc is always used with 0 as the second argument

```
(define specialize-to-0
  (specialize specialize 0))
```

Cool isn't it?

If we subsitute

```
(define specialize-to-0
  (specialize specialize 0))
(define specialize-to-0 (lambda
  (x)(specialize x 0)))
```