Scheme Notes 02

Geoffrey Matthews

Department of Computer Science
Western Washington University

April 13, 2018

Procedures in Scheme

```
(define f
  (lambda (x) (* x x))
  )
```

```
(f 3) => 9
(f 4) => 16

((lambda (x) (* x x)) 3) => 9
((lambda (x) (* x x)) 4) => 16
```

Local Variables in Procedures: Closures

```
(define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      n
    )
  )
)

(counter)   =>  1
(counter)   =>  2
(counter)   =>  3
```

A Procedure that Returns a Procedure

```
(define curry
  (lambda (a)
    (lambda (b)
      (+ a b)
    )
  )
)
```

```
(curry 3)        =>  #<procedure>

((curry 3) 4)    =>  7
```

A Procedure that Returns a Counter

```
(define new-counter
  (lambda ()
    (let ((n 0))
      (lambda ()
        (set! n (+ n 1))
        n)    )  )  )

(define x (new-counter))
(define y (new-counter))

(x)  => 1
(x)  => 2
(x)  => 3
(y)  => 1
(y)  => 2
(x)  => 4
```

Procedures as Returned Values: Derivatives

```
(define d
  (lambda (f)
    (let* ((delta 0.00001)
           (two-delta (* 2 delta)))
      (lambda (x)
        (/ (- (f (+ x delta)) (f (- x delta)))
           two-delta)))))
```

```
((d (lambda (x) (* x x x)))  5)     => ?
((d sin) 0.0)                       => ?
```

Procedures as Returned Values: Procedures as Data

```
(define make-pair
  (lambda (a b)
    (lambda (i)
      (cond ((= i 0) a)
            ((= i 1) b)
            (else (error 'bad-index))))
  ))

(define x (make-pair 4 8))
(define y (make-pair 100 200))
(define z (make-pair x y))

(x 0)        => ?
(y 1)        => ?
((z 1) 0)    => ?
```

Procedures as parameters

Summation notation:

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

In scheme:

```
(define sum
  (lambda (a b f)
    (if (> a b)
        0
        (+ (f a) (sum (+ a 1) b f)))))

(sum 1 10 square) => 385
(sum 1 10 (lambda (x) (* x x x))) => 3025
```

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Because then you have problems with

$$\sum_{i=a}^{b} i^2 = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \boxed{?} = a^2 + \ldots + b^2$$

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Because then you have problems with

$$\sum_{i=a}^{b} i^2 = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \boxed{?} = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \lambda i.i^2 = a^2 + \ldots + b^2$$

Matches Scheme Code Better, Too

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

```
(define sum
  (lambda (a b f)
    (if (> a b)
        0
        (+ (f a) (sum (+ a 1) b f)))))
```

We have the same problem with derivatives

$$\frac{d}{dx} f(x) = f'(x)$$

$$\frac{dx^2}{dx} = 2x$$

$$D(f) = f'$$

$$D(\lambda x.x^2) = \lambda x.2x$$

The chain rule

With pure functions it's easy:

$$D(f \circ g) = (D(f) \circ g) \cdot D(g)$$

With applied functions:

$$\begin{aligned}
F &= f \circ g \\
F(x) &= f(g(x)) \\
F'(x) &= f'(g(x))g'(x) \\
\frac{dF(x)}{dx} &= \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx}
\end{aligned}$$

Or, if we let $z = f(y)$ and $y = g(x)$ (which is weird) then it looks kind of nice and is easy to memorize (cancel fractions!):

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Finding fixed points

x is a *fixed point* of f if $x = f(x)$
For some functions you can find fixed points by iterating:
$x, f(x), f(f(x)), f(f(f(x))), \ldots$

Fixed points in scheme:

```
(define fixed-point
  (lambda (f)
    (let
        ((tolerance 0.0001)
         (max-iterations 10000))
      (letrec
          ((close-enough?
            (lambda (a b) (< (abs (- a b)) tolerance)))
           (try
            (lambda (guess iterations)
              (let ((next (f guess)))
                (cond ((close-enough? guess next) next)
                      ((> iterations max-iterations) #f)
                      (else (try next (+ iterations 1))))))))
           )
        (try 1.0 0)))))

(fixed-point cos)  => 0.7390547907469174
(fixed-point sin)  => 0.08420937654137994
(fixed-point (lambda (x) x)) => 1.0
(fixed-point (lambda (x) (+ x 1))) => #f
```

Remember Newton's Method?

Newton's method:
If y is a guess for $\sqrt{x}$, then the average of y and x/y is an even better guess.

x	guess	quotient	average
2	1.0	2.0	1.5
2	1.5	1.3333333333333333	1.4166666666666665
2	1.4166666666666665	1.411764705882353	1.4142156862745097
2	1.4142156862745097	1.41421143847487	1.4142135623746899

...

Evidently, we want to iterate, and keep recomputing these things until we find a value that's close enough.

Newton's Method Using Functional Programming

```
(define sqrt
  (lambda (x)
    (fixed-point
      (lambda (y)
        (/
           (+ y (/ x y))
           2
        )))))
```

Procedures as Returned Values: Newton's Method Again

```
(define average-damp
  (lambda (f)
    (lambda (x) (/ (+ x (f x)) 2))))

(define sqrt
  (lambda (x)
    (fixed-point (average-damp (lambda (y) (/ x y))))))
```