Higher-Order Procedures

- Today's topics
  - Procedural abstractions
  - Capturing patterns across procedures – Higher Order Procedures

What is procedural abstraction?

Capture a common pattern

(* 2 2)
(* 57 57)
(* k k)

(lambda (x) (* x x))

Actual pattern

Formal parameter for pattern

Give it a name (define square (lambda (x) (* x x)))

Note the type: number → number

Other common patterns

- 1 + 2 + … + 100
- 1 + 4 + 9 + … + 100^2
- 1 + $1/3^2$ + $1/5^2$ + … + $1/101^2$ (≈ $\pi^2/8$)

$$\sum_{k=1}^{100} k \qquad \sum_{k=1}^{100} k^2 \qquad \sum_{k=1,odd}^{101} k^{-2}$$

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))

(define (sum-squares a b)
  (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ 1 a) b))))

(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ 2 a) b))))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```


Let's examine this new procedure

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

What is the type of this procedure?

1. What type is the output?
2. How many arguments?
3. What type is each argument?

Is deducing types mindless, or what?

Higher order procedures

- A higher order procedure:
  takes a procedure as an *argument* or returns one as a *value*

```
(define (sum-integers a b)
      (if (> a b)
          0
          (+ a (sum-integers (+ 1 a) b))))
```

$$\sum_{k=a}^{b} k$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a) (sum term (next a) next b))))

(define (new-sum-integers a b)
 (sum (lambda (x) x) a (lambda (x) (+ 1 x)) b))
```


Higher order procedures

```
(define (sum-squares a b)
     (if (> a b)
         0
         (+ (square a)
            (sum-squares (+ 1 a) b))))
```

$$\sum_{k=a}^{b} k^2$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a) (sum term (next a) next b))))

(define (new-sum-squares a b)
       (sum square a (lambda (x) (+ 1 x)) b))
```

Higher order procedures

```
(define (pi-sum a b)
    (if (> a b)
        0
        (+ (/ 1 (square a))
           (pi-sum (+ a 2) b))))
```

$$\sum_{k=a,odd}^{b} k^{-2} \approx \pi^2/8$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)(sum term (next a) next b))))

(define (new-pi-sum a b)
     (sum (lambda (x) (/ 1 (square x))) a
          (lambda (x) (+ x 2)) b))
```

7/29

Higher order procedures

- Takes a procedure as an argument *or returns one as a value*

```
(define (new-sum-integers a b)
        (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (new-sum-squares a b)
        (sum square a (lambda (x) (+ x 1)) b))

(define (add1 x) (+ x 1))

(define (new-sum-squares a b)(sum square a add1 b))

(define (new-pi-sum a b)
      (sum (lambda (x) (/ 1 (square x))) a
           (lambda (x) (+ x 2)) b))

(define (add2 x) (+ x 2))

(define (new-pi-sum a b)
    (sum (lambda (x) (/ 1 (square x))) a add2 b))
```

8/29

Returning A Procedure As A Value

```
(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))

(define incrementby (lambda (n) . . . ))

(define add1 (incrementby 1))
(define add2 (incrementby 2))
. . .
(define add37.5 (incrementby 37.5))
```

incrementby: number → (number → number)

```
(define (sum term a next b)
  ; type: (num->num),num,(num->num),num -> num
  (if (> a b)
      0
      (+ (term a)(sum term (next a) next b))))
```

9/29

Returning A Procedure As A Value

```
(define incrementby
   ; type: num -> (num->num)
   (lambda(n) [                    ]))

  (incrementby                    2) →
( (lambda(n) (lambda (x) (+ x n))) 2)

       (lambda (x) (+ x 2))

(incrementby 2) → a procedure of one var (x) that
                  increments x by 2

((incrementby 3) 4) → ?
```

10/29

Quick Quiz

```
(define incrementby
      (lambda(n)(lambda (x) (+ x n)))) → undefined

(define f1 (incrementby 6)) →

(f1 4) →

(define f2 (lambda (x)(incrementby 6))) →

(f2 4) →

((f2 4) 6) →
```

11/29

Procedures as values: Derivatives

$f : x \rightarrow x^2$ $\qquad g : x \rightarrow x^3$

$f' : x \rightarrow 2x$ $\qquad g' : x \rightarrow 3x^2$

- Taking the derivative is a higher-order function: $D(f) = f'$
- What is its *type?*

D: (num → num) → (num → num)

12/29

Computing derivatives

- A good approximation:

$$Df(x) \approx \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$$

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                   epsilon)) ))
```

(number → number) → (number → number)

13/29

Using "deriv"

```
(define square (lambda (y) (* y y)) )
(define epsilon 0.001)

((deriv square) 5)
```

```
(define deriv
  (lambda (f)
    (lambda (x) (/ (- (f (+ x epsilon))
                      (f x))
                   epsilon)) ))
```

14/29

Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      `()
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      `()
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))

(define (map proc lst)
  (if (null? lst)
      `()
      (adjoin (proc (first lst))
              (map proc (rest lst)))))
```

Transforms a list to a list, replacing each value by the procedure applied to that value

```
(define (square-list lst)
  (map square lst))
(square-list (list 1 2 3 4)) →

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
(double-list (list 1 2 3 4)) →
```

15/29

Common Pattern #2: Filtering a List

```
(define (filter pred lst)
  (cond ((null? lst) `())
        ((pred (first lst))
         (adjoin (first lst)
                 (filter pred (rest lst))))
        (else (filter pred (rest lst)))))

(filter even? (list 1 2 3 4 5 6))
→ (2 4 6)
```

16/29

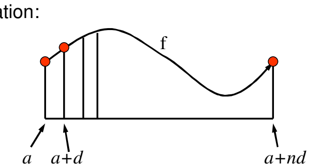
Common Pattern #3: Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
         (add-up (rest lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
         (mult-all (rest lst)))))

(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (first lst)
          (fold-right op init (rest lst)))))

(define (add-up lst)
  (fold-right + 0 lst))
```

17/29

Using common patterns over data structures

- We can more compactly capture our earlier ideas about common patterns using these general procedures.
- Suppose we want to compute a particular kind of summation:



$$\sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta)$$

18/29

Using common patterns over data structures

```
(define (generate-interval a b)
  (if (> a b)
      '()
      (cons a (generate-interval (+ 1 a) b))))
(generate-interval 0 6) →
```

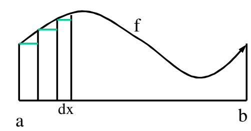
```
(define (sum f a delta n)
 (add-up
       (map (lambda (i) (f (+ a (* i delta))))
            (generate-interval 0 n))))
```

$$\sum_{i=0}^{n} f(a + i\delta)$$

Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))




```
(define (integral f a b)
   (let ((dx (/ (- b a) ni)))
      (* dx (sum f a dx ni))))

(define ni 10000)
```


Computing Integrals

```
(define (integral f a b)
   (let ((delta (/ (- b a) ni)))
      (* (sum f a delta ni) delta)))
(define ni 10000)
```

$$\int_0^a \frac{1}{1+x^2}dx = ?$$

```
(define atan (lambda (a)
  (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```


Finding fixed points of functions

- Square root of x is defined by $\sqrt{x} = x/\sqrt{x}$
- If we think of this as a transformation $f(y) = x/y$ then $\sqrt{x}$ is a **fixed point** of f, i.e. $f(\sqrt{x}) = \sqrt{x}$
- Here's a common way of finding fixed points
  - Given a guess x_1, let new guess be $f(x_1)$
  - Keep computing f of last guess, until close enough

```
(define (close? u v)   (< (abs (- u v)) 0.0001))
(define (fixed-point f i-guess)
     (define (try g)
        (if (close? (f g) g)
            (f g)
            (try (f g))))
    (try i-guess))
```


Using fixed points

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 1)  → 1.6180
```

or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

$$y = \frac{x}{y}$$
$$y^2 = x$$
$$y = \sqrt{x}$$

Unfortunately if we try (sqrt 2), this oscillates between 1, 2, 1, 2,

```
(define (fixed-point f i-guess)
     (define (try g)
        (if (close? (f g) g)
            (f g)
            (try (f g))))
    (try i-guess))
```


So damp out the oscillation

```
(define (average-damp f)
  (lambda (x)
     (average x (f x))))
```

Check out the type:

(number → number) → (number → number)

that is, this takes a procedure as input, and returns a **NEW** procedure as output!!!

```
((average-damp square) 10)
→((lambda (x) (average x (square x))) 10)
→(average 10 (square 10))
→ 55
```

… which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x y)))
    1))
```

- Compare this to Heron's algorithm for sqrt from a previous lecture
  - That was the same process, but the key ideas (**repeated guessing** and **averaging**) were tangled up with the particular code for sqrt.
  - Now the ideas have been abstracted into higher-order procedures, and the sqrt-specific code is just provided as an argument.

```
(define (cube-root x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x (square y))))
    1))
```


Procedures as arguments: a more complex example

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

(compose square double 3)
(square (double 3))
(square (* 3 2))
(square 6)
(* 6 6)
36
```

What is the type of compose? Is it:

```
(number → number), (number → number), number → number
```


Compose works on other types too

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

```
(compose
 (lambda (p) (if p "hi" "bye"))   boolean → string
 (lambda (x) (> x 0))             number → boolean
 -5                               number
 ) →                              result: a string
```

Will any call to compose work?

```
(compose < square 5)

(compose square double "hi")
```


Type of compose

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

- Use type variables.

```
compose:   (B → C), (A → B), A → C
```

- Meaning of type variables:
  All places where a given type variable appears must match when you fill in the actual operand types
- The constraints are:
  - **f** and **g** must be functions of one argument
  - the argument type of **g** matches the type of **x**
  - the argument type of **f** matches the result type of **g**
  - the result type of **compose** is the result type of **f**

Higher order procedures

- Procedures may be passed in as arguments
- Procedures may be returned as values
- Procedures may be used as parts of data structures

- Procedures are first class objects in Scheme!!