Higher-Order Procedures

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

What is procedural abstraction?

Capture a common pattern

\( \lambda (x) \times x \times x \)

\( \lambda (x) \times 2 \times 2 \)

\( \lambda (x) \times 3 \times 3 \)

\( \lambda (x) \times k \times k \)

Formal parameter for pattern

Give it a name

\( \text{(define square} \ (\lambda (x) \times x \times x)) \)

Note the type: number \(\rightarrow\) number

Other common patterns

- \( 1 + 2 + \ldots + 100 \)
- \( 1 + 4 + 9 + \ldots + 100 \)
- \( 1 + \frac{1}{3} + \frac{1}{5} + \ldots + \frac{1}{101} \)
- \( \sum_{k=1}^{100} k \)
- \( \sum_{k=1}^{100} k^2 \)

Let’s examine this new procedure

\( \text{(define} \ (\text{sum}\ \text{term}\ a\ \text{next}\ b) \)

\( \text{(if} \ (> a b) \)

\( 0 \)

\( (+ (\text{term} a) (\text{sum}\ \text{term}\ (\text{next}\ a)\ \text{next}\ b))) \)

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

\( \text{(define} \ (\text{sum-integers}\ a\ b) \)

\( \text{(if} \ (> a b) \)

\( 0 \)

\( (+ (\text{sum-integers} \ a \ 1\ b))))) \)

\( \text{(define} \ (\text{sum-squares}\ a\ b) \)

\( \text{(if} \ (> a b) \)

\( 0 \)

\( (+ (\text{square}\ a) \ (\text{sum-squares} \ (+ 1\ a)\ b))))) \)

\( \text{(define} \ (\text{new-sum-integers}\ a\ b) \)

\( \text{(sum}\ (\lambda (x) \ x) a \ (\lambda (x) \ (+ 1\ x) b))) \)

Higher order procedures

\( \text{(define} \ (\text{sum-squares}\ a\ b) \)

\( \text{(if} \ (> a b) \)

\( 0 \)

\( (+ (\text{square}\ a) \ (\text{sum-squares} \ (+ 1\ a)\ b))) \)

\( \text{(define} \ (\text{new-sum-squares}\ a\ b) \)

\( \text{(sum}\ \text{(sum-squares}\ a\ b) \ (\lambda (x) \ (+ 1\ x) b))) \)

\( \text{(define} \ (\text{pi-sum}\ a\ b) \)

\( \text{(if} \ (> a b) \)

\( 0 \)

\( (+ (\text{pi-sum}\ (+ 2\ a)\ b))) \)

\( \sum_{k=1}^{\infty} \frac{1}{k^2} \)
Higher order procedures

\[\sum_{k=1}^{k/2} \frac{1}{x^2} = \pi^2/6\]

\[7/29\]

\[\sum_{k=1}^{k/2} \frac{1}{x^2} = \pi^2/6\]

\[8/29\]

Higher order procedures

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

\[9/29\]

\[\sum_{k=1}^{k/2} \frac{1}{x^2} = \pi^2/6\]

\[10/29\]

\[\sum_{k=1}^{k/2} \frac{1}{x^2} = \pi^2/6\]

Returning A Procedure As A Value

\[11/29\]

\[\sum_{k=1}^{k/2} \frac{1}{x^2} = \pi^2/6\]

Procedures as values: Derivatives

\[12/29\]

\[13/29\]

Quick Quiz

\[14/29\]

\[15/29\]
Computing derivatives

- A good approximation:

\[ Df(x) = \frac{f(x + \epsilon) - f(x)}{\epsilon} \]

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

Using “deriv”

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

((deriv square) 5)

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))

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)

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)))))

(fold-right + 0 lst)

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:

\[ \sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta) \]
Using common patterns over data structures

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

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

Integration as a procedure

```
Integration under a curve f is given roughly by
\[ \int_a^b f(x) \, dx = \sum_{i=0}^{n-1} f(a + i \cdot \delta) \delta \]
```

Computing Integrals

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

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) \rightarrow 1.6180
```

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

So damp out the oscillation

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

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

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

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

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

```
((average-damp square) 10)
((lambda (x) (average x (square x))) 10)
(average 10 (square 10))
```
... 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)

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))))

(type variables,
compose: (B → C), (A → B), A → C

• 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!!