

## Higher-order procedures

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

## Another sum

```
(define (sum-of-even-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
          (sum-of-even-cubes
           (+ a 2)
           b)))))
```

## Generalized sum

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

## Examples

```
(define (inc x) (+ x 1))
(define (inc-by-2 x) (+ x 2))
sum-of-squares:
(sum square 1 inc 10)
sum-of-even-cubes:
(sum cube 2 inc-by-2 10)
```

## Terminology

- Procedures that accept other procedures as input or return a procedure as output are **higher-order procedures**.
- The other procedures are **first-order procedures**.

## Using higher-order procedures

```
(midpoint rule)
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f
        (+ a (/ dx 2.0))
        add-dx
        b)
     dx))
```

## Evaluation

- `(integral square 0 1 0.001)` returns `0.33333325`
- Exact answer is  $1/3$

## One more

```
(define (identity x) x)
sum of consecutive integers:
(sum identity 1 inc 10)
```

## Procedures without names

- `(lambda (<param1> <param2> . . . )  
 <body>)`
- `(define (square x) (* x x))`
- `(define square  
 (lambda (x) (* x x)))`
- `lambda = create-procedure`

## Procedures are first-class objects

- Can be the value of variables
- Can be passed as parameters
- Can be return values of functions
- Can be included in data structures

## Using nameless procedures

- `(sum (lambda (x) x)  
 1  
 (lambda (x) (+ x 1))  
 10)`
- `(sum (lambda (x) (* x x x))  
 2  
 (lambda (x) (+ x 2))  
 10)`

## (continued)

- `(define (integral f a b dx)  
 (* (sum f  
 (+ a (/ dx 2.0))  
 (lambda (x) (+ x dx))  
 b)  
 dx))`

## More local variables

```
((lambda (x y) (+ (* x x)
                  (* y y))))
5
7)
((lambda (v1 v2 ...) <body>)
 val-for-v1
 val-for-v2
 ...)
```

## Another example

- To compute  $(x + y) + (x + y)^2$   
`(define (f x y)`  
    `((lambda (z) (+ z (* z z)))`  
      `(+ x y)))`

## The let special form ...

```
(let ((<var1> <expr1>)
      (<var2> <expr2>)
      ...
      <body>)
```

## translates to ...

```
((lambda (<var1> <var2> ...) <body>)
 <expr1>
 <expr2>
 ...)
```

## Using let

```
(define (f x y)
  (let ((z (+ x y)))
    (+ z (* z z))))
```

## General methods

- Example: half-interval-method for finding root of equation  $f(x) = 0$
- Assume  $f$  is continuous,  $f(a) < 0$  and  $f(b) > 0$
- Strategy: evaluate  $f$  at midpoint; decide which half-interval brackets a root; repeat

## Auxiliary functions

```
(define (avg x y) (/ (+ x y) 2))
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

## The basic search

```
(define (search f np pp)
  (let ((mp (avg np pp)))
    (if (close-enough? np pp)
        mp
        (let ((tv (f mp)))
          (cond ((positive? tv)
                 (search f np mp))
                ((negative? tv)
                 (search f mp pp))
                (else mp))))))
```

## The main procedure

```
(define (half-interval-method f a b)
  (let ((va (f a))
        (vb (f b)))
    (cond ((and (negative? va) (positive? vb))
           (search f a b))
          ((and (positive? va) (negative? vb))
           (search f b a))
          (else (error "values have same sign"
                       a
                       b))))))
```

## (h-i-m continued)

To find root of  $x^2 + x - 1 = 0$ , try  
(half-interval-method  
  (lambda (x) (+ (\* x x) x -1))  
  0.0  
  1.0)  
  
0.61767578125

## Fixed point of a function

- A **fixed point** of a function  $f$  is a point  $x$  where  $f(x) = x$ .
- To find a fixed point, start with any value  $x$ , then compute the sequence  $f(x)$ ,  $f(f(x))$ ,  $f(f(f(x)))$ , . . . The limiting value (if it exists) is a fixed point.
- In practice, stop when successive values are close enough to each other.

## Fixed-point operator

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? x y)
    (< (abs (- x y)) tolerance))
  (define (try-guess)
    (let ((next-guess (f guess)))
      (if (close-enough? guess next-guess)
          next-guess
          (try next-guess))))
  (try first-guess))
```

## Some close calls

```
(fixed-point (lambda (x) (+ 1 (sin x)))) 1.0)
--> 1.93456...
(fixed-point square 0.5) --> 0
(fixed-point square 1.0) --> 1
(fixed-point square 2.0) --> infinity
```

## Square-root as a fixed point

- Consider  $f(y) = x/y$ . then if  $f(y) = y$ , we have  $x/y = y$  or  $x = y^2$ , so  $\text{sqrt}(x)$  is fixed-point of  $f$ .
- (define (sqrt x)  
 (fixed-point (lambda (y)  
 (/ x y))  
 1.0))
- Doesn't work!

## Conservative guesses

- Instead of going all the way from  $y$  to  $f(y)$ , be conservative and only go part of the way
- For continuous functions, small changes in  $y$  will produce small changes in  $f(y)$
- Technique is called **damping**
- For sqrt function, use average of  $y$  and  $x/y$  (average damping)

## A better version

```
(define (sqrt x)
  (fixed-point
   (lambda (y)
     (average y (/ x y)))
   1.0))
```