Scheme procedures are "first class"

- Procedures can be manipulated like the other data types in Scheme
  - A variable can have a value that is a procedure
  - A procedure value can be passed as an argument to another procedure
  - A procedure value can be returned as the result of another procedure
  - A procedure value can be included in a data structure

```scheme
(define area-of-disk (lambda (r) (* pi (* r r))))
```

Special form: `lambda`

- `(lambda (formals) (body))`
- A lambda expression evaluates to a procedure
  - It evaluates to a procedure that will later be applied to some arguments producing a result
- `(formals)`
  - Formal argument list that the procedure expects
- `(body)`
  - Sequence of one or more expressions
  - The value of the last expression is the value returned when the procedure is actually called

"Define and name" with lambda

```scheme
(define area-of-disk
  (lambda (r)
    (* pi (* r r))))
```
"Define and use" with lambda

- \((\lambda (r) (\times \pi r r))\)

Separating procedures from names

- We can treat procedures as regular data items, just like numbers
  - and procedures are more powerful because they express behavior, not just state
- We can write procedures that operate on other procedures - applicative programming

```scheme
(define (min-fx-gx f g x)
  (min (f x) (g x)))

(define (identity x) x)
(define (square x) (* x x))
(define (cube x) (* x x x))

(define (min-fx-gx f g x)
  (min (f x) (g x)))

(min-fx-gx square cube 2)         ; (min 4 8) => 4
(min-fx-gx square cube -2)        ; (min 4 -8) => -8
(min-fx-gx identity cube 2)       ; (min 2 8) => 2
(min-fx-gx identity cube (/ 1 2)) ; (min 1/2 1/8) => 1/8
```

```scheme
(define (s-fx-gx s f g x)
  ; define a procedure 's-fx-gx' that takes:
  ; s - a combining function that expects two numeric arguments
  ; f, g - two functions that take a single numeric argument and
  ; return a single numeric value f(x) or g(x)
  ; x - the point at which to evaluate f(x) and g(x)
  ; s-fx-gx returns s(f(x),g(x))

(s-fx-gx min square cube 2) ; => (min 4 8) = 4
(s-fx-gx min square cube -2) ; => (min 4 -8) = -8
(s-fx-gx + square cube 2) ; => (+ 2 8) = 12
(s-fx-gx - cube square 3) ; => (- 8 9) = 18
```

Exercises

1. Define a procedure 's-fx-gx' that takes:
   - s - a combining function that expects two numeric arguments
   - f, g - two functions that take a single numeric argument and
   - return a single numeric value f(x) or g(x)
   - x - the point at which to evaluate f(x) and g(x)
   - s-fx-gx returns s(f(x),g(x))

(define (s-fx-gx s f g x)
Exercises

2. Define a procedure `positive-fx-gx?` that takes:
   - \( f, g \) - two functions that take a single numeric argument and return a single numeric value \( f(x) \) or \( g(x) \)
   - \( x \) - the point at which to evaluate \( f(x) \) and \( g(x) \)
   `positive-fx-gx?` returns true if \( f(x) \) and \( g(x) \) are both > 0, false otherwise

3. Define a procedure `compose` that takes:
   - \( f, g \) - two functions that take a single numeric argument and return a single numeric value \( f(x) \) or \( g(x) \)
   `compose` returns a function that accepts one numeric argument and returns \( f(g(x)) \)

4. (CHALLENGE) Define a procedure `apply-n-times` that takes:
   - \( f \) - a function that takes a single numeric argument and returns a single numeric value \( f(x) \)
   - \( n \) - the number of times to apply the function \( f \)
   `apply-n-times` returns a function that accepts a numeric argument \( x \) and the result of applying \( f() \) to \( x \), \( n \) times

Example: summation

We can always define specific functions for specific applications

\[
\sum_{i=a}^{b} i^3
\]

Example: summation

\[
\sum_{i=a}^{b} \frac{1}{i}
\]

Generalize?

Where can we generalize to perhaps provide broader application?

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

```
(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

General purpose sum

Define the sum function so that it takes functions as arguments that calculate the current term and the next index:

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

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

```
(define (pi-sum a b)
  (if (> a b)
    0
    (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```
Redefine sum-cubes using `sum`

```scheme
(define (inc i) (+ i 1))
(define (cube x) (* x x x))
(define (sum-cubes a b)
  (define (sum term a next b)
    (if (> a b) 0
      (+ (term a) (sum term (next a) next b))))

(sum-cubes a b)
)```

Redefine pi-sum using `sum`

```scheme
(define (pi-term i)
  (/ 1.0 (* i (+ i 2))))
(define (pi-next i)
  (+ i 4))
(define (pi-sum a b)
  (sum pi-term a pi-next b))

(pi-sum a b)
)```

Redefine pi-sum using `sum` and `lambda`

```scheme
(define (pi-sum2 a b)
  (sum
    (lambda (i) (/ 1.0 (* i (+ i 2))))
    a
    (lambda (i) (+ i 4))
    b))

(pi-sum2 a b)
)```

Redefine pi-sum using `sum` and lambda

```scheme
(define (pi-sum3 a b)
  (sum
    (lambda (i) (/ 1.0 (* i (+ i 2))))
    a
    (lambda (i) (+ i 4))
    b))

(pi-sum3 a b)
)```

-a numeric interval-

```latex
\begin{align*}
  x &= [0:.1:1] \\
  y &= \sin(x) \\
  \text{plot}(x, y)
\end{align*}
```

Calculate-h

```scheme
; define a function to calculate an interval size (b-a)/n
(define calculate-h (lambda (a b n) (/ (- b a) n)))

; try it out on [0,1]
(calculate-h 0 1 10)
)```

Anonymous calculate-h

```scheme
; do the same thing without naming the function
((lambda (a b n) (/ (- b a) n)) 0 1 10)
)```
calculate last-x

; define a function that figures out what the beginning
; of the last interval is
; calculate a+(n-1)*h directly
(define (last-x1 a b n)
  (+ a (* (- n 1) (/ (- b a) n))))

last-x using a helper function

; calculate a+(k*h) using a simple function, and
; pre-calculate k and h to pass to the function
(define (last-x2 a b n)
  (define (use-kh k h)
    (+ a (* k h )))
  (use-kh (- n 1) (/ (- b a) n)))

last-x using anonymous helper function

; calculate a+(k*h) using an anonymous function
(define (last-x3 a b n)
  ((lambda (k h) (+ a (* k h )))
   (- n 1)
   (/ (- b a) n)))

last-x with concealed anonymous function

; hide the use of the anonymous function by using let
(define (last-x4 a b n)
  (let ((h (/ (- b a) n))
       (k (- n 1)))
    (+ a (* k h ))))

Special form let

(let ((var1) (exp1))
 (var2) (exp2))
<body>

- When the let is evaluated, each expression exp_i is evaluated and the resulting value is associated with the related name var_i, then the body is evaluated.
- There is no order implied in the evaluation of exp_i

scope and let

; an example in scoping with let
(define x 2)
(let ((x 3)
      (y (+ x 2)))
 (* x y))
((lambda (x y)
    (* x y))
  3
  (+ x 2))
nesting lets lets us get x

(define x 2)
(let ((x 3))
  (let ((y (+ x 2)))
    (* x y))
(let* ((x 3) (y (+ x 2)))
  (* x y))

Special form let*

(let* ((\(\langle var_1\rangle\langle exp_1\rangle\))
        (\(\langle var_2\rangle\langle exp_2\rangle\)))
  \(\langle body\rangle\))

- When the let* is evaluated, each expression \(exp_i\) is evaluated in turn and the resulting value is associated with the related name \(var_i\), then the body is evaluated.
- The \(exp_i\) are evaluated in left to right order
  » each binding indicated by \(\langle var_i\rangle\langle exp_i\rangle\) is part of the environment for \(\langle var_{i+1}\rangle\langle exp_{i+1}\rangle\) and following
  » This is exactly equivalent to nesting the let statements

an iterator with parameter h

(define (show-x1 a b n)
  (define (iter h count)
    (if (> count n)
        (newline)
        (begin
          (display (+ a (* h count)))
          (display " ")
          (iter h (+ count 1)))))
  (iter (/ (- b a) n) 0))

h defined in enclosing scope

(define (show-x2 a b n)
  (let ((h (/ (- b a) n)))
    (define (iter count)
      (if (> count n)
          (newline)
          (begin
            (display (+ a (* h count)))
            (display " ")
            (iter (+ count 1)))))
    (iter 0)))

Special form begin

(begin \(\langle exp_1\rangle\langle exp_2\rangle\ldots\langle exp_n\rangle\))

- Evaluate the \(exp_i\) in sequence from left to right
- The value returned by the entire begin expression is the value of \(exp_n\)
- Best used to sequence side effects like I/O
  » for example displaying each of the x values in show-x
- There is implicit sequencing in the body of a lambda procedure or a let but we generally don’t use it
  » the procedure returns the value of the last \(exp_i\), so the body of most of our procedures consists of one expression only

show-x

Welcome to DrScheme, version 205.
Language: Standard (R5RS).
> (show-x2 0 1 10)
 0 1/10 1/5 3/10 2/5 1/2 3/5 7/10 4/5 9/10 1 4/5