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

define and name

(define (area-of-disk 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

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

"Define and use" with lambda

• ((lambda (r) (* pi r r)) 1)
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))
```

- Let's apply `min-fx-gx`.

```scheme
(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
```

Exercises

- 4. Define a procedure `apply-n-times` that takes:
  - `f` - a function that takes a single numeric argument and return a single numeric value `f(x)`
  - `n` - the number of times to apply the function `f`
  - `apply-n-times` returns a function that accepts one numeric argument `x` and the result of applying `f()` to `x`, `n` times.
  - Example: `(apply-n-times square 2) 3)`

```scheme
(define (apply-n-times f n x)
  (let loop ((n n) (result x))
    (if (= n 0) result
        (loop (- n 1) (f result))))))
```