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

Recall: Define and name a procedure

- \((\text{define } \langle \text{name} \rangle \langle \text{formal params} \rangle \langle \text{body} \rangle)\)
  - \text{define} - special form
  - \text{name} - the name that the procedure is bound to
  - \text{formal params} - names used within the body of procedure
  - \text{body} - expression (or sequence of expressions) that will be evaluated when the procedure is called.
  - The result of the last expression in the body will be returned as the result of the procedure call
define and name

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

- define a procedure that takes one argument \( r \) and calculates \( \pi \cdot r^2 \)
- bind that procedure to the name \texttt{area-of-disk}

- The name of the variable that holds the procedure and the actual body of the procedure are separate issues

Special form: \texttt{lambda}

- \texttt{(lambda \langle\text{formals}\rangle \langle\text{body}\rangle)}
  - A lambda expression evaluates to a procedure
    - it evaluates to a procedure that will later be applied to some arguments producing a result
  - \langle\text{formals}\rangle
    - formal argument list that the procedure expects
  - \langle\text{body}\rangle
    - 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 a procedure that takes one argument \( r \) and calculates \( \pi \cdot r^2 \)
- bind that procedure to the name \texttt{area-of-disk}

- The name of the variable that holds the procedure and the actual body of the procedure are separate issues

"Define and use" with lambda

((lambda (r) (* pi r r)) 1)

- define a procedure that takes one argument \( r \) and calculates \( \pi \cdot r^2 \)
- apply that procedure to the argument value 1
- return the result \( \pi \)

- The body of the procedure is applied directly to the argument and is never named at all
Separating procedures from names

- We can now start to treat procedures as regular data items, just like numbers
  - and procedures are more powerful because they express behavior, not just state
- We can now write procedures that operate on other procedures - applicative programming
  - higher order functions
  - functions that take functions as arguments and do standard things with them

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

```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
```

```scheme
; define s-fx-gx
(define s-fx-gx
  ; define a procedure that takes:
  ; s - a combining function that expects two numeric arguments
  ; and returns a single numeric value
  ; 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))
  (lambda (f g x)
    (s (f x) (g x))))
```

```scheme
(define identity
  (lambda (x) x))

(define square
  (lambda (x) (* x x)))

(define cube
  (lambda (x) (* x x x)))

(define (s-fx-gx s f g x)
  (s (f x) (g x)))
```
apply s-fx-gx

\begin{align*}
(s-fx-gx \text{ min square cube } 2) & \Rightarrow (\text{min 4 8}) = 4 \\
(s-fx-gx \text{ min square cube } -2) & \Rightarrow (\text{min 4 -8}) = -8 \\
(s-fx-gx + \text{ square cube } 2) & \Rightarrow (+ 2 8) = 12 \\
(s-fx-gx - \text{ cube square } 3) & \Rightarrow (- 27 9) = 18
\end{align*}

\begin{itemize}
\item Example : summation
\end{itemize}

We can always define specific functions for specific applications

\begin{align*}
(\text{define (sum-cubes a b)}) \\
\quad (\text{if (> a b)}) \\
\quad \quad 0 \\
\quad \quad (+ (\text{cube a}) (\text{sum-cubes (+ a 1) b})))
\end{align*}

\begin{align*}
(\text{define (pi-sum a b)}) \\
\quad (\text{if (> a b)}) \\
\quad \quad 0 \\
\quad \quad (+ (/ 1.0 (* a (+ a 2))) (\text{pi-sum (+ a 4) b})))
\end{align*}

\begin{itemize}
\item Generalize?
\end{itemize}

Where can we generalize to perhaps provide broader application?

\begin{align*}
(\text{define (sum-cubes a b)}) \\
\quad (\text{if (> a b)}) \\
\quad \quad 0 \\
\quad \quad (+ (\text{cube a}) (\text{sum-cubes (+ a 1) b})))
\end{align*}
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 a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))
```

Redefine sum-cubes using sum

```
(define (inc i) (+ i 1))

(define (cube x) (* x x x))

(define (sum-cubes a b)
  (sum (cube a b)
       a
       b))
```

Redefine pi-sum using sum

```
(define (pi-term i)
  (/ 1.0 (* i (+ i 2))))

(define (pi-next i)
  (+ i 4))

(define (pi-sum a b)
  (sum pi-term a b))
```

Redefine pi-sum using sum and lambda

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

- Define a procedure `double`
  » takes a procedure of one argument as its argument
    ie, a procedure `f` that can be applied `(f x)`
  » returns a procedure that applies the original procedure twice
    ie, a procedure `g` that is `(f (f x))`
- For example
  » `(double inc)` returns a procedure `(inc (inc x))`

The parts of `double`

```
(define (double f)
  (lambda (z)
    (f (f z))))
```

`double` takes one argument, a function `f`

`double` returns a function

```
(define (double f)
  (lambda (z)
    (f (f z))))
```

the function that `double` returns takes one argument

```
(define (double f)
  (lambda (z)
    (f (f z))))
```

the function that `double` returns is `f` composed with itself

Evaluate expressions with `double`

```
(define (double f)
  (lambda (z)
    (f (f z))))

(define (inc x) (+ x 1))

(define (plus4 x) ((double (double inc)) x))

((double inc) 3) ; 2 + 3 = 5

(plus4 10) ; 10 + 4 = 14
```