What is that **lambda** thing?
Formulating Abstractions with Higher Order Procedures
*Structure and Interpretation of Computer Programs*  
*Autumn Term, 2006*
Formulating Abstractions with Higher-Order Procedures

Idea: procedures identify common patterns

\[
(* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))))
(* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))))))
\]

\[===>\]

\[
(define \text{factorial} \ n) \\
\text{(if} \ (= \ n \ 0) \\
\text{1} \\
(* \ n \ (\text{factorial} \ (- \ n \ 1))))
\]
How can we identify, abstract, and represent "higher order" common features?

\[
(+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))))))
\]
\[
= \sum k \\
0 \leq k \leq 8
\]

\[
(+ 8^2 (+ 7^2 (+ 6^2 (+ 5^2 (+ 4^2 (+ 3^2 (+ 2^2 (+ 1^2 0^2))))))))
\]
\[
= \sum k^2 \\
0 \leq k \leq 8
\]

\[
(+ (1/8^2) (+ (1/7^2) (+ (1/6^2) (+ (1/5^2) \\
(+ (1/4^2) (+ (1/3^2) (+ (1/2^2) (+ (1/1^2) (1/0^2))))))))
\]
\[
= \sum 1/(k^2) \\
0 \leq k \leq 8
\]

All of these have the pattern

\[
= \sum f(k) \\
a \leq k \leq b
\]
In Scheme, formal parameters of procedures can be bound to procedures as well as numbers, so that we can represent such common patterns:

\[
\begin{align*}
&\text{(define (sum term a next b)} \\
&\quad (\text{if (> a b) } \\
&\qquad 0 \\
&\qquad (+ (term a) \\
&\qquad \quad (\text{sum term (next a) next b}))))
\end{align*}
\]

Example of use:

\[
\begin{align*}
&\text{(define (sum-squares a b)} \\
&\quad (\text{sum square a 1+ b})
\end{align*}
\]

[NB: (define (1+ x) (+ 1 x)) ... 
... but the unary 1+ is built into Scheme]
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))

(define (sum-squares a b) (sum square a 1+ b))

Substitution model:

(sum-squares 3 5)
(sum square 3 1+ 5)
(if (> 3 5) 0 (+ (square 3) (sum square (1+ 3) 1+ 5)))
(+ (square 3) (sum square (1+ 3) 1+ 5))
...
(+ 9 (sum square (1+ 3) 1+ 5))
(+ 9 (sum square 4 1+ 5))
...
(+ 9 (+ 16 (sum square 5 1+ 5)))
...
(+ 9 (+ 16 (+ 25 (sum square 6 1+ 5))))
...
(+ 9 (+ 16 (+ 25 (if (> 6 5) 0 (+ (square 6) (sum square (1+ 6) 1+ 5))))))
...
(+ 9 (+ 16 (+ 25 0)))
...
50
An infinite series for pi/8:

\[
\frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \frac{1}{9 \times 11} + \frac{1}{13 \times 15} + \ldots = \frac{\pi}{8}
\]

Scheme code to sum the first \( n \) terms of the series:

Note the (first) multiplier in the \( n \)-th term is \( 4n-3 \)

\[
\begin{align*}
&\text{(define (funny-square x) (* x (+ x 2)))} \\
&(\text{define (term n)} \\
&\quad (/ 1 (funny-square (- (* 4 n) 3)))) \\
&(\text{define (pi-sum n)} \\
&\quad (sum term 1 1+ n))
\end{align*}
\]
Definite integrals

(define (integral fn a b dx)
  (* dx
   (sum fn
    (+ a (/ dx 2))
    (lambda (x) (+ x dx))
    b))))

Question: what's this funny lambda thing? Alternative:

(define (integral fn a b dx)
  (define (increment x) (+ x dx))
  (* dx
   (sum fn
    (+ a (/ dx 2))
    increment
    b))))

When we write (define (square x) (* x x)) we are specifying:
(1) a procedure that multiplies its argument by itself, and
(2) a name for that procedure (why not call it Bob, Chuck, or Dave?)

Alternatively:

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

Then (lambda (x) (* x x)) evaluates to the (nameless) procedure, which we then name using define. In the code for integral, we give the procedure without giving a name to it.

Why would you name something? Because you use it more than once.
(define (integral fn a b dx)
  (* dx
       (sum fn
        (+ a (/ dx 2))
        (lambda (x) (+ x dx))
        b)))

Now use integral as a building block:

\[
\text{arctangent } a = \int_{0}^{a} \frac{dx}{1 + x^2}
\]

(define (arctangent a)
  (integral (lambda (x) (/ 1 (1+ (square x))))
            0
            a
            .001))
A further, useful abstraction

(define (accumulate combiner null-value term a next b)  
  (if (> a b)  
      null-value  
      (combiner (term a)  
                  (accumulate combiner null-value term (next a) next b)))

(define (sum term a next b)  
  (accumulate + 0 term a next b))

(define (factorial n)  
  (accumulate * 1 (lambda (x) x) 1 1+ n))

Expand the body of factorial using the substitution model (NB: a, b not substituted for):

(if (> a b)  
  1  
  (* ((lambda (x) x) a) (accumulate * 1 (lambda (x) x) (1+ a) 1+ b)))

What does the procedure (lambda (x) x) do?
Square roots, revisited.

A fixed point of a function f(-) is an input a such that f(a)=a
To compute the square root of x, find the fixed point of

\[ f(y) = \frac{y + \frac{x}{y}}{2} \]

(define (fixed-point fn guess)
  (if (good-enough? fn guess)
      guess
      (fixed-point fn (fn guess))))

(define (good-enough? fn guess)
  (< (abs (- (fn guess) guess)) .0001))

(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1))
(define (fixed-point fn guess)
  (if (good-enough? fn guess)
      guess
      (fixed-point fn (fn guess))))

Note that procedure **fixed-point** does not always evaluate to something!
Substitution model:

(fixed-point 1+ 2)
(fixed-point 1+ 3)
...

Solve for \(x\):

\[8x + 3 = 15 + 6x\]

Or: \(x = (12 + 6x)/8\)  \[a \text{ recursive definition}??\]

(define (f x) (/ (+ 12 (* 6 x)) 8))

(fixed-point f 1)
;Value: 5.999996224522291

Is there an easier way to solve this equation?
Digression to crazy but true idea: Recursion is about solving equations!

Exercise: solve the following equations:

\[ 3x^2 - 5x + 4 = 6 \]

\[
\begin{align*}
\text{(define } & (g \ x) \ (\text{sqrt } ((+ \ 2 \ (* \ 5 \ x)) \ 3))) \\
\text{(fixed-point g 1)}
\end{align*}
\]

;Value: 1.999998974260502

\[ x = 10 \quad \text{What is } (\text{fixed-point } (\text{lambda } (x) \ 10) \ 1) ? \]

\[ f(x) = f(-x) \]

\[ 3x^2f''(x) + 2xf'(x) - 10x^2f(x) + 4x^3 = 0 \]

\[ f = (\text{lambda } (x) \ (* \ x \ x)) \]

\[ f = (\text{lambda } (n) \ (\text{if } (= \ n \ 0) \ 1 \ (* \ n \ (f (- n \ 1))))) \]
A fish weighs 20 pounds plus half its own weight---how much does the fish weigh?

\[
f = 20 + \frac{f}{2} \\
= 20 + \frac{1}{2} (20 + \frac{f}{2}) \\
= (1+ \frac{1}{2})20 + \frac{f}{4} \\
= (1+ \frac{1}{2})20 + \frac{1}{4}(20 + \frac{f}{2}) \\
= (1+ \frac{1}{2} + \frac{1}{4})20 + \frac{f}{8}
\]
Derivatives (from calculus, not finance...)

```
(define (deriv fn x dx)
  (/ (- (fn (+ x dx)) (fn x))
      dx))
```

This computes the derivative of a function at a point -- but what if what we want to compute is the derivative function?

```
(define (deriv fn dx)
  (lambda (x)
    (/ (- (fn (+ x dx)) (fn x))
       dx))

;Value deriv
```

This procedure returns another procedure as a value!

```
(define dsquare (deriv square .00001))
;Value: dsquare

(dsquare 2)
;Value: 4.000010000027032

(((deriv square .00001) 5)
 ;Value: 10.0000099999444615
```
Derivatives and *higher-order procedures*:

```
(define (deriv fn dx)
    (lambda (x)
        (/ (- (fn (+ x dx)) (fn x))
            dx))

;Value deriv

This procedure returns *another procedure* as a value!

(define dsquare (deriv square .00001))
;Value: dsquare

Substitution model:

(deriv square .00001)
(lamba (x) (/ (- (square (+ x .00001)) (square x)) .00001))

(dsquare 2)
(/ (- (square (+ 2 .00001)) (square 2)) .00001))

;Value: 4.000010000027032

((deriv square .00001) 5)
;Value: 10.000009999444615
```
Composing two functions:

(define (compose f g)
  (lambda (x) (f (g x))))
;Value: compose

((compose square 1+) 7)
;Value: 64

(define (repeated fn n)
  (if (= n 1)
      fn
      (compose fn (repeated fn (- n 1))))))
;Value: repeated

((repeated square 3) 5)
;Value: 390625
Composing two functions:

(define (compose f g)
  (lambda (x) (f (g x))))
;Value: compose

((compose square 1+) 7)
;Value: 64

Substitution model:

((lambda (x) (square (1+ x))) 7)
(square (1+ 7))
Substitution model:

(define (repeated fn n)
  (if (= n 1)
      fn
      (compose fn (repeated fn (- n 1)))))

((repeated square 3) 5)
  ((if (= 3 1)
    square
    (compose square (repeated square (- 3 1)))) 5)
...
  ((compose square (compose square (repeated square 2))) 5)
...
  ((compose square (compose square (repeated square 1))) 5)
...
  ((compose square (compose square square)) 5)
  ((compose square (lambda (x) (square (square x)))) 5)
  ((lambda (y) (square
    ((lambda (x) (square (square x))) y))) 5)
  (square ((lambda (x) (square (square x))) 5))
  (square (square (square 5)))
...
390625
Notice how ((repeated square n) k) expands proportional to n.
A space-efficient alternative function composition:

(define (repeated fn n)
  (if (= n 1)
      fn
      (lambda (x) ((repeated fn (- n 1)) (fn x))))))

Notice that in the substitution model, we do not "evaluate under lambda"---that is, don't simplify the body of a lambda expression until the lambda expression is applied.

(((repeated square 3) 5)
  ...
  ((lambda (x) ((repeated square (- 3 1)) (square x))) 5)
  ((repeated square (- 3 1)) (square 5))
  ((repeated square 2) 25)
  ...
  ((lambda (x) ((repeated square (- 2 1)) (square x))) 25)
  ((repeated square (- 2 1)) (square 25))
  ...
  (square (square 25))
  ...
  (square 625)
  ...
  390625

The above process uses space constant in \( n \), not linear in \( n \) as before.
Currying: single versus multiargument functions (a useful programming idiom...)

(+ 8 5) ;Value: 13

(define (plus n)
  (lambda (x) (+ n x))) ;Value: plus

(define 2+ (plus 2)) ;Value: 2+

(2+ 5) ;Value: 7

((plus 8) 5) ;Value: 13

Analyzing functionality

+: (Int * Int) --> Int
plus: Int --> (Int --> Int)
\[+: (\text{Int} \times \text{Int}) \rightarrow \text{Int}\]
\[\text{plus}: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})\]

\[
\begin{align*}
\text{(define (curry fn)} \\
&\quad (\lambda (x) \quad \text{(lambda (y) \quad (fn x y))))) \\
;\text{Value: curry}
\end{align*}
\]

\[
\begin{align*}
\text{(define plus (curry +))} \\
;\text{Value: plus}
\end{align*}
\]

**Substitution model:**

\[
\begin{align*}
\text{(((curry +) 8) 5)} \\
\text{(((lambda (x) (lambda (y) (+ x y)) 8) 5)} \\
\text{((lambda (y) (+ 8 y)) 5)} \\
\text{(+ 8 5)} \\
\text{13}
\end{align*}
\]

**Reverse "uncurrying"**

\[
\begin{align*}
\text{(define (uncurry curried-fn)} \\
&\quad (\lambda (x y) (((curried-fn x) y)))
\end{align*}
\]

**Exercise:** Check that + and (uncurry (curry +)) do the same thing...
Yet another crazy idea:
Programs are proofs, and types are theorems

\[ f : A \rightarrow B \]

- `f` is a procedure turning input of type `A` into output of type `B`
- `f` is a proof of theorem `A --> B`, in other words, a construction that turns any proof of `A` into a proof of `B`

\[ x : A \]

\[ (f \ x) : B \]

- Function application is *modus ponens* from logic!
Higher order procedures and functionality --
What kind (type) of answers are supposed to be computed?

This is the key to solving the first problem set.
We write informal "types" to describe functionality.

type Height= Real
type Velocity= Real
type Fuel= Real
type Shipstate= Height*Velocity*Fuel  
   [a 3-field record...]
type Fuel-burnrate= Real
type Strategy= Shipstate --> Fuel-burnrate

Problem 4:

random-choice: (Strategy * Strategy) --> Strategy

Note
(Strategy * Strategy) --> Strategy =
   ((Shipstate --> Fuel-burnrate) * (Shipstate --> Fuel-burnrate)) -->
   (Shipstate --> Fuel-burnrate)

...a real higher-order procedure!...
Question: what kind of "random choice" is this? Once, or continually?
type Height= Real
type Velocity= Real
type Fuel= Real

\text{type Shipstate=} \text{Height*Velocity*Fuel} \quad \text{[a 3-field record...]}\text{)}
\text{type Fuel-burnrate= Real}
\text{type Strategy= Shipstate --> Fuel-burnrate}

Problem 5:

\text{height-choice: (Strategy * Strategy * Int) --> Strategy}

Problem 6:

\text{choice: (Strategy * Strategy * (Shipstate --> Boolean)) --> Strategy}
Procedures are **first-class** values:
(a concept introduced by Christopher Strachey)

They can be named by variables.
They can be parameters of procedures.
They can be returned as results of procedures.

What else is first-class in Scheme?
Integers? **define**?
Some very useful *syntactic sugar*: `let`

\[
f(x,y) = (x^2 + y^2)(x^2 - y^2)
\]

\[
\text{(define (fn x y)}
\text{\quad (* (+ (square x) (square y))}
\text{\quad (- (square x) (square y))))}
\]

Better: *(or, why is it better? or is it?)*

\[
\text{(define (fn x y)}
\text{\quad (define (helper a b) (* (+ a b) (- a b)))}
\text{\quad (helper (square x) (square y))))}
\]

Why name something used only once?

\[
\text{(define (fn x y)}
\text{\quad ((lambda (a b) (* (+ a b) (- a b))) (square x) (square y))))}
\]

Hard to read, so instead use this *syntactic sugar*:

\[
\text{(define (fn x y)}
\text{\quad (let ((a (square x))}
\text{\quad \quad (b (square y)))}
\text{\quad (* (+ a b) (- a b)))}}
\]
\( (\text{let } ((v_1 \ e_1) \ (v_2 \ e_2) \ ... \ (v_n \ e_n)) \ b) \Rightarrow ((\text{lambda } (v_1 \ v_2 \ ... \ v_n) \ b) \ e_1 \ e_2 \ ... \ e_n) \)

Pitfall: Dependence of variable definitions...
(Variables in above are "bound" simultaneously, not sequentially, so they cannot depend on each other.)

(define x 2); Value: x

(let ((x 3) (y (+ x 2))) (* x y)); Value: 12

where does the underlined \( x \) get its value?

Rewrite the last expression as:

\( (((\text{lambda } (x \ y) (* x y)) \ 3) \ (+ \ \underline{x} \ 2)) \)

It is a "free" (not bound) variable

Compare with

(let ((x 3))
  (let ((y (+ x 2)))
    (* x y))
); Value: 15

An analogy: how much is 2 and 2?

Math nerd: "4"
Engineer: "3.99998"
Lawyer: "How much would you like it to be?"

[Programming language design is like being the lawyer... what do we want the meaning to be?]

Using the substitution model, rewrite the last expression as:

\( (((\text{lambda } (x) (\text{let } ((y (+ x 2))) (* x y))) \ 3)\)
\( (((\text{lambda } (x) ((\text{lambda } (y) (* x y)) (+ x 2))) \ 3)\)
\( (((\text{lambda } (y) (* 3 y)) (+ 3 2)))\)
\( (((\text{lambda } (y) (* 3 y)) 5)\)
\( (* 3 5)\)

15