Higher Order Functions

Recall that a higher-order function either
- takes operators as parameters,
- yields an operator as its result,
- or both

Why do we need higher-order function?
- In mathematics, not all operators deal exclusively with numbers
  - +, -, *, /, expt, log, mod, ...
  - Take in numbers, return numbers
- But mathematical operations like ? (e.g., ? i=0:n; (i=3)
  - Take in operators
  - Return operators (or numbers)
Operators as Parameters

\[
\begin{align*}
\text{(sum (lambda (x) (* x x)) 0 5)} & \quad x^2 \quad a \\
\text{(+ (lambda (x) (* x x)) 0)} & \\
\text{(sum (lambda (x) (* x x)) 1 5)} & \quad x \quad 0 \\
\text{(+ 0 (+ (lambda (x) (* x x)) 1)} & \quad \text{(sum (lambda (x) (* x x)) 2 5))} \\
\text{...} & \\
\text{(+ 0 (+ 1 (+ 4 (+ 9 (+ 16 (+ 25 0))))))} & 
\end{align*}
\]

Generalized summation

\[
\text{high} \quad \text{low} \\
\begin{align*}
\text{high} & \\
\text{?} & \\
\text{f(x)} & \\
\text{x} & \text{low} \\
\text{What if we don't want to go up by 1?} & \\
\text{Supply another procedure} & \\
\text{given current value, finds the next one} & \\
\text{(define (gsum f low next high)} & \\
\text{(if (> low high) 0} & \\
\text{(+ (f low)} & \\
\text{(gsum f (next low) next} & \\
\text{high)))))} & 
\end{align*}
\]

stepping by 1, 2, ...

\[
\begin{align*}
\text{(define (step1 n) (+ n 1))} & \\
\text{(define (sum f low high) (gsum f low step1 high))} & \\
\text{(define (step2 n) (+ n 2))} & \\
\text{(define (sum2 f low high) (gsum f low step2 high))} & 
\end{align*}
\]
stepping by 2

\[
\begin{align*}
\text{(define (square n) (* n n))} \\
\text{(sum square 2 4)} = 2^2 + 3^2 + 4^2 \\
\text{(sum2 square 2 4)} = 2^2 + 4^2 \\
\text{(sum2 (lambda (n) (* n n n)) 1 10)} = 1^3 + 3^3 + 5^3 + 7^3 + 9^3
\end{align*}
\]

Using lambda

\[
\begin{align*}
\text{(define (step2 n) (+ n 2))} \\
\text{(define (sum2 f low high)} \\
\quad \text{(gsum f low step2 high))}
\end{align*}
\]

Why not just write this as:

\[
\begin{align*}
\text{(define (sum2 f low high)} \\
\quad \text{(gsum f low (lambda (n) (+ n 2)) high))}
\end{align*}
\]

Don’t need to name tiny one-shot functions

Using lambda

\[
\begin{align*}
\text{How about:} \\
\text{sum of n\textsuperscript{4} for n = 1 to 100, stepping by 5?} \\
\text{(gsum (lambda (n) (* n n n n))} \\
\quad \text{1} \\
\quad \text{(lambda (n) (+ n 5))} \\
\quad \text{100})
\end{align*}
\]

Note: the n’s in the lambdas are independent of each other
One last function

\[ \int_{a}^{b} f(x) \, dx \]

- Definite integral of \( f(x) \) from \( a \) to \( b \)
- If this was a sum, we could do it...

approximate as a sum

\[ \int_{a}^{b} f(x) \, dx \approx \frac{b-a}{n} \left[ f(a) + f(a+\frac{b-a}{n}) + \ldots + f(b) \right] \]

Integration in scheme...

\[ \int_{a}^{b} f(x) \, dx \approx \frac{b-a}{n} \left[ f(a) + f(a+\frac{b-a}{n}) + \ldots + f(b) \right] \]

(define (integral f a b dx)
  (* dx
    (gsum f a (lambda (x) (+ x dx)) b)))
Example

\[
\int_0^1 x^2 \, dx = \frac{1}{3}
\]

\[
\text{(integral (lambda (x) (* x x)) 0 1 0.0001)} \Rightarrow 0.3333833349999416
\]

Operators as return values

- The derivative operator
  - Takes in… \( f(x) \) \( \frac{df}{dx}(F(x)) \)
  - A function
  - Returns… another function

- The integration operator
  - Takes in… \( F(x) \) \( \int f(x) \, dx \)
  - A function from numbers to numbers, and
  - A value of the function at some point
  - E.g. \( F(0) = 0 \)
  - Returns… another function from numbers to numbers

Further motivation

- Besides mathematical operations that inherently return operators, it’s nice to have operations that help construct larger, more complex operations.

- Example:

  (define add1 (lambda (x) (+ x 1)))
  (define add2 (lambda (x) (+ x 2)))
  (define add3 (lambda (x) (+ x 3)))
  (define add4 (lambda (x) (+ x 4)))
  (define add5 (lambda (x) (+ x 5)))

- Repetitive and tedious
- Is there a way to abstract this?
Abstract “Up”

- Generalize to a function that can create adders:

  (define (make-addn n)
   (lambda (x) (+ x n)))

- Equivalent definition

  (define make-addn
   (lambda (n)
     (lambda (x) (+ x n)))))

How do I use it?

- (define (make-addn n)
    (lambda (x) (+ x n)))

- (define add3 (make-addn 3))

- (add3 4)
  7

Evaluating...

- (define add3 (make-addn 3))
- Evaluate (make-addn 3)
  - Evaluate 3 -> 3.
  - Evaluate make-addn ->
    - (lambda (n) (lambda (x) (+ x n)))
  - Apply make-addn to 3...
    - Substitute 3 for n in (lambda (x) (+ x n))
    - Get (lambda (x) (+ x x))
  - Make association:
    - add3 bound to (lambda (x) (+ x x))
Evaluating...

- `(add3 4)`
  - Evaluate 4 -> 4
  - Evaluate add3
    - `(lambda (x) (+ x 3))`
  - Apply `(lambda (x) (+ x 3))` to 4
    - Substitute 4 for x in `(+ x 3)`
    - `(+ 4 3)`
    - 7

make-addn’s “signature”

```scheme
(define (make-addn n)
  (lambda (x) (+ x n)))
```
- Takes in a numeric argument n
- Returns a function…
  - …which has, within it, a value “pre-substituted” for n.
- Notice: Standard substitution model still works

Evaluate a function call

To Evaluate a function call…
1. Evaluate the arguments
2. Apply the function

To Apply a function call…
1. Replace the function argument variables with the values given in the call everywhere they occur
2. Evaluate the resulting expression
Clarify Substitution Model

Replace the function argument variable (e.g., “n”) with the value given in the call everywhere it occurs.

There is an exception:

Do not substitute for the variable inside any nested lambda expression that also uses the same variable as one of its arguments.

Example

(define weird-protection-example
  (lambda (n)
    (lambda (n) (+ n n))))

What is the output for (define foo (weird-protection-example 3))?

Should bind foo to (lambda (n) (+ n n))

...not (lambda (n) (+ 3 3))

Variable Definition/Substitution

Intuitively:

(lambda (n) EXPR)

Acts as a shield

> Protects any “n”’s inside EXPR from substitution
> “n” bounces off
> Everything else gets through (this guard)

Formally:

Substitution of a variable is based on tightest enclosing lambda where variable is an argument.
### Another Example

```scheme
(define select-op
  (lambda (b)
    (if b
      (lambda (a b)
        (and a b))
      (lambda (a b)
        (or a b))))
)
```

? `(select-op #t)`
- `(lambda (a b) (and a b))`

? `(select-op #f)`
- `(lambda (a b) (or a b))`

### Summation Problem Revisited

```scheme
(define (sum f low high)
  (if (> low high) 0
      (+ (f low)
          (sum f (+ low 1) high))))
)
```

Abstract the high value:
```scheme
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))
)
```

### To use...

```scheme
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))
)
```

```scheme
(define squares-to-n
  (make-sum (lambda (x) (* x x)) 0))
```

### Example

```
(define select-op
  (lambda (b)
    (if b
      (lambda (a b)
        (and a b))
      (lambda (a b)
        (or a b))))
)
```

? `(select-op #t)`
- `(lambda (a b) (and a b))`

? `(select-op #f)`
- `(lambda (a b) (or a b))`

```scheme
(define (sum f low high)
  (if (> low high) 0
      (+ (f low)
          (sum f (+ low 1) high))))
)
```

Abstract the high value:
```scheme
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))
)
```

```scheme
(define squares-to-n
  (make-sum (lambda (x) (* x x)) 0))
```

To use...
```scheme
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))
)
```

```scheme
(define squares-to-n
  (make-sum (lambda (x) (* x x)) 0))
```

```
```
Result

(define (make-sum f low)
  (lambda (high)
    (sum f low high)))

(define squares-to-n
  (make-sum (lambda (x) (* x x)) 0))

squares-to-n ends up bound to:
  (lambda (high)
    (sum (lambda (x) (* x x)) 0 high)))

Calling defined function

(squares-to-n 5)
...
(sum (lambda (x) (* x x)) 0 5)
...
55

Higher Order Functions

Functions that return
  numbers...
  abs, square, sum

Functions that return
  functions that return numbers
  make-addr, make-sum

Function that return
  functions that return functions that return numbers....
  ...etc.
Simple multiple-abstract-up

\[
\begin{align*}
\text{(define make-2stage-add} & \text{ (lambda (a)} \\
& \text{ (lambda (b)} \\
& \text{ (lambda (c)} \\
& \text{ (+ a b c))))) \\
\end{align*}
\]

Using make-2stage-add

\[
\begin{align*}
\text{(define make-2stage-add} & \text{ (lambda (a)} \\
& \text{ (lambda (b)} \\
& \text{ (lambda (c)} \\
& \text{ (+ a b c))))) \\
\text{(define make-add3n} & \text{ (make-2stage-add 3))} \\
\text{make-add3n gets bound to:} & \text{ (lambda (b)} \\
& \text{ (lambda (c) (+ 3 b c)))} \\
\end{align*}
\]

Using make-add3n

\[
\begin{align*}
\text{make-add3n gets bound to:} & \text{ (lambda (b)} \\
& \text{ (lambda (c) (+ 3 b c)))} \\
\text{(define add34} & \text{ (make-add3n 4))} \\
\text{add34 gets bound to:} & \text{ (lambda (c) (+ 3 4 c)))} \\
\end{align*}
\]
Using add34

- add34 bound to: (lambda (c) (+ 3 4 c))
- (add34 5)
- (+ 3 4 5)
- 12

Higher Order Functions

- Functions as arguments
  - (define (sum f low high) (+ (f low) (sum ...)
- Functions as return values
  - (define (make-addn n) (lambda (x) (+ x n)))
- Functions defined in terms of functions
  - (define (make-sum f low) (lambda (high) (sum f low high)))
- Also derivative, integral...
- Functions which return functions which return functions
  - make-2stage-add

Variables

- Scheme's variables have lexical scope
- Global variables:
  - (define x 10)
- Parameters of lambda are examples of local variables:
  - They get bound each time the procedure is called, and their scope is that procedure's body
  - (define x 9)
  - (define add2 (lambda (x) (+ x 2)))
  - x => 9
  - (add2 3) => 5
  - (add2 x) => 11
  - x => 9
### Set!

Set! modifies the lexical binding of a variable.

(set! x 20)

- modifies the global binding of x from 9 to 20.
- If the set! was inside add2's body, it would have modified the local x.

```scheme
(define add2
  (lambda (x) (set! x (+ x 2)))
  x)
```

- The set! here adds 2 to the local variable x, and returns that value.
- We can call add2 on the global x, as before:
  - (add2 x) => 22 (Remember global x is now 20, not 9!)
- The set! inside add2 affects only the local variable used by add2. The global x is unaffected by the set! to the local x.
  - x => 20 (Global x remains unchanged)

---

### Global Variables

What will the following code produce?

```scheme
(define counter 0)
(define bump-counter
  (lambda ()
    (set! counter (+ counter 1))
    counter))
```

- bump-counter is a zero-argument procedure (also called a thunk).
- Each time it is called, it modifies the global variable counter - it increments it by 1 - and returns its current value.
  - (bump-counter) => 1
  - (bump-counter) => 2
  - (bump-counter) => 3

---

### Local Variables

**Old way:**

```scheme
(define (foo x y)
  (define z (+ (* x x) (* y y)))
  (sqrt (* (- z x) (- z y))))
```

**New way:**

```scheme
(define (foo x y)
  (let ((z (+ (* x x) (* y y))))
    (sqrt (* (- z x) (- z y))))
```

---
Let

- Local variables can be introduced without explicitly creating a procedure.
  (let ((x 1)
        (y 2)
        (z 3))
   (list x y z))

- will output the list (1 2 3)
- As with lambda, within the let-body, the local x (bound to 1) shadows the global x (which is bound to 20).

Let is lambda in disguise

- (let ((var1 <expr1>)
       (var2 <expr2>))
   <body>)

- ((lambda (var1 var2) <body>)
  <expr1> <expr2>)

- substitution model is unchanged!

Compound Data Types

- Compound data types are built by combining values from other data types
- Strings
  - Strings are sequences of characters (not to be confused with symbols, which are simple data that have a sequence of characters as their name)
  - You can specify strings by enclosing the constituent characters in double quotes
  - Strings evaluate to themselves
    "Hello, World!"
    => "Hello, World!"
- The procedure string takes a bunch of characters and returns the string made from them:
  (string #\h #\e #\l #\l #\o)
  => "hello"
Compound Data Types

- **Vectors**
  - Vectors are sequences like strings, but their elements can be anything, not just characters.
  - The elements can be vectors themselves, which is a good way to generate multidimensional vectors.
  - Here's a way to create a vector of the first five integers:
    
    (vector 0 1 2 3 4)
    
    => #(0 1 2 3 4)

- Note Scheme's representation of a vector value: a # character followed by the vector's contents enclosed in parentheses.

Data Abstraction: Motivation

- Consider a complex number
  - \( X = a + bi \)
  - How do we represent and use?
  - For motivation, let's just use what we know…
  - Need both numbers (a=real) and (b=imaginary) for each complex number
  - Define some operations…
    - add
    - subtract
    - negate
    - multiply
    - magnitude

Defining Complex-add

- (define (complex-add ar ai br bi) …)
- What does it return?
  - Can only return a number.
  - But we need two results…
  - So... have to write separate functions?!
Complex-add

- Return real part
  (define (complex-add-real ar ai br bi)
      (+ ar br))

- Return imaginary part
  (define (complex-add-imag ar ai br bi)
      (+ ai bi))

Reminder: Complex-Multiply

- (ar+i) + (bi+i)
  = (ar*br + ai*bi) + (br*ai + ar*bi)
  = (ar*br - ai*bi) + (ar*bi + br*ai)i

  (define (complex-multiply-real ar ai br bi)
      (- (* ar br) (* ai bi)))

  (define (complex-multiply-imag ar ai br bi)
      (+ (* ar bi) (* ai br)))

More Complex Operations

- foo(A,B,C) = A+B*C
  (define (foo-real ar ai br bi cr ci)
      (complex-add-real ar ai
          (complex-multiply-real br bi cr ci)
          (complex-multiply-imag br bi cr ci)))

  (define (foo-imag ar ai br bi cr ci)
      (complex-add-imag ar ai
          (complex-multiply-real br bi cr ci)
          (complex-multiply-imag br bi cr ci)))

  Have to handle every component separately.
  - Makes it hard to understand.
  - Exposes lots of underlying complexity.
  - Have to return each component separately.
  - Have to write separate operations.
  - (Perform lots of operations redundantly)
We need...

foo(A,B,C) = A+B*C

- ...some way to bundle together numbers.
- Then we can treat them as a single item.
- Ideally should be able to write:
  (define (d a b c)
    (complex-add a (complex-multiply b c)))
- We need a way to compose things.

Compound Data

- Attaching Operation: cons
  - takes two data items
  - Creates a composite data item
  - ...that contains the original two

  E.g.
  (cons 3 4)
  :Value: (3 . 4)

Box and Pointer Diagrams

Numbers

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

(cons 3 4)
Three ways to describe

1. With code…
   • (cons 3 4)
2. With the way Scheme displays…
   • (3 . 4)
3. With diagrams…
   
   ![Diagram]

Decomposing

- Can retrieve components with two operations:
  - car
    - Returns the first item
  - cdr
    - Returns the second item

- (define a (cons 3 4))
- (car a)
  3
- (cdr a)
  4

Box and Pointer Diagrams

```
"cons cell"

"car pointer"

"cdr pointer"

3 4
```
Box and Pointer Diagrams

(define a (cons 3 4))

Note

- cons works on all types:
  - (cons 3 4)
  - (cons 'a 'b)
  - (cons (lambda (x) x) (lambda (x) (/ 1 x)))
  - (cons (cons 3 4) (cons 4 5))

Cons arguments

- Cons does not require homogeneous arguments
- These are all valid:
  - (cons 3 '#t)
  - (cons 3 (lambda (x) (+ 3 x)))
  - (cons 3 (cons 4 (lambda (x) x)))
Complex Numbers Revisited

Define an abstraction of a complex number:
(\text{define (make-complex real imaginary)}
\text{(cons real imaginary))}
(\text{define (get-real complex) (car complex))}
(\text{define (get-imag complex) (cdr complex))}

Complex Numbers: operators

Define our operations in terms of this abstraction:
(\text{define (complex-add a b)}
\text{(make-complex}
\text{ (+ (get-real a) (get-real b))}
\text{ (+ (get-imag a) (get-imag b))))}

Sample Evaluation

(\text{define a (make-complex 3 4))}
\text{a gets (3 . 4)}
(\text{define b (make-complex 1 2))}
\text{b gets (1 . 2)}
(\text{complex-add a b)}
\text{(make-complex (+ (get-real (3 . 4)) (get-real (1 . 2)))}
\text{(+ (get-imag (3 . 4)) (get-imag (1 . 2))))}
Continuing Evaluation

- (make-complex (+ (get-real (3 . 4)) (get-real (1 . 2)))
  (+ (get-imag (3 . 4)) (get-imag (1 . 2))))
- (make-complex (+ (car (3 . 4)) (car (1 . 2)))
  (+ (cdr (3 . 4)) (cdr (1 . 2))))
- (make-complex (+ 3 1)
  (+ 4 2))
- (cons 4 6)
  (4 . 6)

Note:

- Cannot type dotted notation into Scheme interpreter (yet…)
  (4 . 6)
- "bad syntax: illegal use of "."
- Need to use cons
  (cons 4 6)
  Dotted pair is a notational convenience

Complex Numbers: multiply

(define (complex-multiply a b)
  (make-complex
   (- (* (get-real a) (get-real b))
      (* (get-imag a) (get-imag b))
      (* (get-real a) (get-imag b)))
   (+ (* (get-real a) (get-imag b))
      (* (get-imag a) (get-real b)))))
Revisiting foo

\[ \text{foo}(A, B, C) = A + B^*C \]

\[
(\text{define (foo a b c)}
  (\text{complex-add a})
  (\text{complex-multiply b c}))
\]

Comparison

\[
(\text{define (foo-real ar ai br bi cr ci)}
  (\text{complex-add-real ar ai})
  (\text{complex-multiply-real br bi cr ci}))
\]

\[
(\text{define (foo-imag ar ai br bi cr ci)}
  (\text{complex-add-imag ar ai})
  (\text{complex-multiply-imag br bi cr ci}))
\]

\[
(\text{define (foo a b c)}
  (\text{complex-add a})
  (\text{complex-multiply b c}))
\]

Details hidden – we get to just focus on operations at this level.