Lecture 5 — August 17, 2015

- Today:
  - Environment encapsulation
  - More higher-order procedures: Fixed points
  - Language typing
  - Dotted-tail notation
- Readings:
  - Finish *SICP* 2.2–2.4
  - Read *SICP* 2.5, 3.1, 3.2

CSE130, Summer Session II

Lambdas encapsulate their environment

A lambda remembers the values of the variables in its environment:

```
(define (make-adder a)
  (lambda (n) (+ n a)))

(define successor (make-adder 1))
(define add-5 (make-adder 5))

(successor 2)
(add-5 10)
(add-5 (add-5 (successor 24)))
```

The **cons-car-cdr** Conspiracy

```
(car (cons a b)) → a
(cdr (cons a b)) → b
```

A group of functions can “conspire” with each other. What we want is to define our own **cons**, **car**, and **cdr** functions:

```
(define (cons a b)
  (lambda (selector)
    (if selector a b)))

(define (car p) (p #t))

(define (cdr p) (p #f))
```

The **`cons-car-cdr`** Conspiracy

- Section 2.1.3 “mind boggling” definition
- We don’t need Booleans, **`cond`**, nor **`if`**!

```
(define (cons a b)
  (lambda (selector)
    (selector a b)))

(define (car p)
  (p (lambda (a b) a)))

(define (cdr p)
  (p (lambda (a b) b)))
```

Fixed points

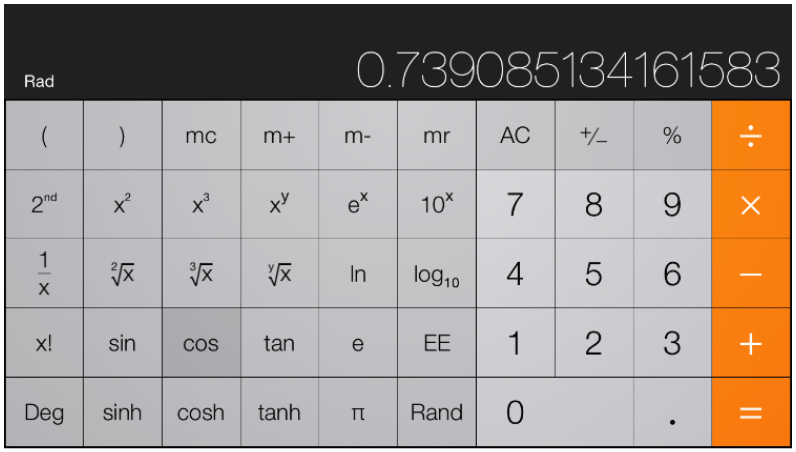
- A number x is called a ***fixed point*** of a function f if x satisfies the equation $f(x) = x$.
  - E.g.,: $\sin(0) = 0, \cos(0.73908513321) = 0.73908513321$
  - E.g.,: $\text{abs}(x)$ for all non-negative x
- For some functions f we can locate a fixed point by making an initial guess and applying f repeatedly…

$f(x)$

$f(f(x))$

$f(f(f(x)))$

$f(f(f(f(x))))$ …



- …until the value doesn't change by some small number

Finding fixed points

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (let loop ([guess first-guess])
    (let ([next (f guess)])
      (if (< (abs (- guess next)) tolerance)
          next
          (loop next)))))
```

- Apply **`f`** to the **`guess`** to get **`next`**
  - if $|guess - next| < tolerance$, you've found it
  - Otherwise, keep on looking with **`next`** as the new **`guess`**

Square roots

- The ***square root*** of n is the x such that $x^2 = n$
  - In other words, an x such that $x = n / x$
  - E.g., the square root of 5 is the x where $x = 5 / x$

- So, let $f(x) = 5 / x$

- Thus, we want the x such that $x = f(x)$
- In other words, we want to find the fixed point of this f

Square roots

- Our first attempt…

```
(define (sqrt n)
  (fixed-point (lambda (x) (/ n x)) 1.0))
```

  doesn't converge.

- E.g., `(sqrt 5)` will just keep repeating 1 and then 5 and then 1… as the guesses:
  - $f(1) = 5$
  - $f(f(1)) = f(5) = 1$

Square roots, continued

- Solution: Don't let the guesses change so much. Instead, try the *average* of the previous guess with the new guess
- This leads to finding the fixed point of a different function:

```
(define (average x y)
  (/ (+ x y) 2))

(define (sqrt n)
  (fixed-point
   (lambda (x) (average x (/ n x)))
   1.0))
```

```
> (sqrt 5)
2.236067977499978
> (* 2.236067977499978 2.236067977499978)
5.000000000000843
```

Square roots, continued

This ***average dampening technique*** can be useful for other cases. Abstract it and we get a procedure that takes a procedure as input and returns a new procedure!

```
; (number -> number) -> (number -> number)
(define (average-damp f)
  (lambda (x)
    (average x (f x))))

; number -> number
(define (sqrt n)
  (fixed-point
   (average-damp
    (lambda (x) (/ n x)))
   1.0))
```

Debugging the process

- Use the output functions **`display`** and **`newline`**:

```
(define (fixed-point f first-guess)
  (let loop ([guess first-guess])
    (display guess)
    (newline)
    (let ([next (f guess)])
      (if (< (abs (- guess next)) tolerance)
          next
          (loop next)))))
```

```
> (sqrt 5)
1.0
3.0
2.3333333333333335
2.238095238095238
2.2360688956433634
2.236067977499978
```

compose

- This higher-order procedure takes two procedures as input:

```
; (number -> number), (number -> number), number -> number
(define (compose f g x)
  (f (g x)))
```

```
(compose square double 3)
(square (double 3))
(square (* 2 3))
(square 6)
(* 6 6)
36
```

compose

- But the body of **compose** itself doesn't require **x** to be a number, nor **f** and **g** to be functions on numbers!

```
(define (compose f g x)
  (f (g x)))
```

```
(compose
 (lambda (p) (if p "yin" "yang"))
 (lambda (x) (> x 0))
 -5)
⟶ "yang"
```

- What is the type of **f**? **g**? **x**? the result?

compose

- But not every combination of functions can be composed:

```
> (compose < square 5)
<: arity mismatch;
 the expected number of arguments does not match the
given number
  expected: at least 2
  given: 1
  arguments.:
```

```
> (compose square double "tofu")
+: contract violation
  expected: number?
  given: "tofu"
  argument position: 1st
  other arguments.:
```

Typing **compose**

```
(define (compose f g x) (f (g x)))
```

- **x** can be of any type, let's call that type *C*
- This is known as a ***type variable:***
  - All places where a given type variable appears must match when you fill in the actual operand types
- Constraints:
  - **f** and **g** must be functions of one argument
  - the argument type of **g** matches the type of **x** (what we call *C*)
  - the argument type of **f** matches the result type of **g** (call it *A*)
  - the result type of **compose** is the result type of **f** (call it *B*)
- So:
  compose: (A -> B), (C -> A), C -> B

“cader” and “caduder”

Common to see in some Lisp code, so you should be aware of it:

```
(define (cadr lst) (compose car cdr lst))
(define (caddr lst) (compose cadr cdr lst))
```

Alternative formulation:

```
; o: (B -> C), (A -> B) -> (A -> C)
(define (o f g)
  (lambda (x) (compose f g x)))

(define cadr (o car cdr))

(define caddr (o car (o cdr cdr)))
```

The **`for-each`** procedure

- A useful higher-order expression for its side effects (e.g., I/O) instead of the value it returns:

```
(define (for-each proc lst)
  (if (not (null? lst))
      (begin
        (proc (car lst))
        (for-each proc (cdr lst)))))
```

```
> (for-each (lambda (x) (display x) (newline))
            (list 57 321 88))
57
321
88
```

- This implementation demonstrates the “one-armed-if” and **`begin`** special forms.
- In Racket, there is no one-armed-if, so just supply some value

Static typing

- With ***static typing***, the type of every expression can be known by analyzing the program
  - Under ***type checking***, the types are explicitly supplied by the programmer (e.g., Java)
  - Under ***type inference***, the types can be deduced based on how expressions are ***used***.
- Type inference in ML:
  - if the + operator is used on a, then a must be an integer
  - if b is initialized by the result of f, then it has the same type of what f returns
  - If c is passed as the first argument to g, then it has the same type as g's first parameter
- Lambdas in Java 8 employ type inference

Dynamic typing

- Under ***dynamic*** or ***latent typing***, the types of (some) expressions can only be determined by executing specific instances of them at runtime
  - In some implementations, every value has a "tag" associated with it, indicating its type (e.g., Scheme)
  - Programmer does not need to specify types
  - When an operation is applied, the tags of its operands are checked to see if the operation is allowed.
  - Might only be caught on a primitive procedure, if the user-defined procedure doesn't check
- You can write this in Scheme, but not in ML:

```
(define (foo n)
  (if (< n 0) (- n) foo))
```

Languages without typing

- An ***untyped language*** offers no safety
  - E.g. in Assembly a floating-point number can be added to an address
  - C/C++ are statically typed, but you can subvert the type system explicitly (type casting) and implicitly (dangling pointers)
- When something "goes wrong" in an untyped language all bets are off: whatever happens is not defined and it depends upon the OS/hardware
- Both statically typed and dynamically typed languages can be ***type safe*** by fully defining the semantics for all scenarios

- Note: The phrases "strong typed" and "weak typed" are meaningless in the sense there are multiple and conflicting definitions for them.

Dotted-tail notation

- Procedures like `+` and `list` have ***variable arity****:* they can take different numbers of arguments
- You can put a dot '`.`' before the final parameter name in a procedure definition to be a list that will be a list of remaining arguments:

```
(define (same-parity first . rest)
  ...)
```

```
> (same-parity 2 3 4 5 6 7 8)
'(2 4 6 8)
> (same-parity 1 2 3 4 5 6 7 8 9 10 11 12)
'(1 3 5 7 9 11)
```

Using dotted-tail notation

```
(define (same-parity first . rest)
  (let iter ([lst (cons first rest)])
    (cond
      [(null? lst) '()]
      [(equal? (even? first) (even? (car lst)))
       (cons (car lst) (iter (cdr lst)))]
      [else (iter (cdr lst))])))
```

Dotted-tail notation with **`lambda`**

- Works for lambdas too, but there must be another parameter before the dot:

```
(define same-parity
  (lambda (first . rest) ...))     ;; OK

(define print-all
  (lambda (. lst) ...))   ;; illegal use of '.'
```

- But you can handle this case by omitting the parenthesis:

```
(define print-all
  (lambda lst
    (let iter ([lst lst])
      (if (not (null? lst))
          (begin
            (display (car lst))
            (newline)
            (iter (cdr lst)))
          'done))))
```

The `list` procedure

```
> (list) ⟶ '()
> (list 1 2 3) ⟶ '(1 2 3)
> (list 4 (list 25 16)) ⟶ '(4 (25 16))
```

- Can this only be implemented with a special form?

```
(define (list . x)

)
```

A problem with recursive calls

- What if we want to call ourselves directly?

```
(define (print-all . lst)
  (if (not (null? lst))
      (begin
        (display (car lst))
        (newline)
        (print-all (cdr lst)))
      'done))
```

- What happens?

```
(print-all 1 2 3)
```

Adding indirection

- One fix is to add indirection, like the original version:

```
(define (print-all . lst)
  (let loop ([lst lst])
    (if (not (null? lst))
        (begin
          (display (car lst))
          (newline)
          (loop (cdr lst)))
        'done)))
```

```
> (print-all 1 2 3)
1
2
3
```

Using **`apply`**

- We want the elements of the list to be the direct arguments, and not have the list *itself* be a single argument.
- The **`apply`** procedure takes a procedure and a list and applies procedure to the elements of the list as arguments:

```
> (apply + (list 1 2 3 4))
                          → 10
> (apply < (list 4 5))
                          → #t
> (apply list (list 2 3))
                          → '(2 3)
```

Calling recursively via **apply**

- Still tail-recursive, because **apply** is the last thing it does!

```
(define (print-all . lst)
  (if (not (null? lst))
      (begin
        (display (car lst))
        (newline)
        (apply print-all (cdr lst)))
      'done))
```

```
> (print-all 1 2 3)
1
2
3  ;; Still Works!
```