



CS1 Recitation

Week 4

Functions that Return Functions

- Write a function that takes values for a , b , c and returns a function that computes the quadratic equation $f(x) = ax^2 + bx + c$

- Arguments: a , b , c
- Result: a function of one argument

- Contract:

```
;; make-quadratic: number number number -> (number -> number)
```

- Code:

```
(define (make-quadratic a b c)  
  (lambda (x) (+ (* a x x) (* b x) c)))
```

- Desugared form:

```
(define make-quadratic  
  (lambda (a b c)  
    (lambda (x) (+ (* a x x) (* b x) c))))
```

make-quadratic

- Evaluate `(define f (make-quadratic 2 -3 5))`
 - `define` is a special form. Evaluate 2nd argument and bind to 1st.
 - Evaluate `(make-quadratic 2 -3 5)`
 - `2` → 2, `-3` → -3, `5` → 5 (yawn)
 - `make-quadratic` →
`(lambda (a b c)`
 `(lambda (x) (+ (* a x x) (* b x) c)))`
 - Apply `(lambda (a b c) ...)` to 2, -3, 5
 - Substitute `a = 2`, `b = -3`, `c = 5` into body of `(lambda (a b c) ...)` →
`(lambda (x) (+ (* 2 x x) (* -3 x) 5))`
 - Evaluate `(lambda (x) (+ (* 2 x x) (* -3 x) 5))` → itself
 - Bind `(lambda (x) ...)` to name `f`

No lambda shielding here!
No value being substituted into inner lambda has the name `x`.

Nested Lambdas

- From class:

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

- Evaluate (`select-op #t`)

- `#t` → `#t`

- `select-op` → `(lambda (b) ...)`

- Apply `(lambda (b) ...)` to `#t`

- Substitute `b = #t` in

- ```
(if b (lambda (a b) (and a b))
 (lambda (a b) (or a b)))
```

- ???

# Lambda Shielding

- A clarification of the substitution model:
  - If nested lambda has an argument with *the same name* as the value being substituted, don't substitute that value into body of nested lambda.
    - Example: substitute  $b = \#t$  into

```
(if b (lambda (a b) (and a b))
 (lambda (a b) (or a b)))
```
  - If value being substituted doesn't have same name as nested lambda's arguments, substitute as normal.
    - Example: substitute  $a = 2, b = -3, c = 5$  into

```
(lambda (x) (+ (* a x x) (* b x) c))
```

# Nested Lambdas (2)

- Evaluate (`select-op #t`)

- `#t` → `#t`

- `select-op` → `(lambda (b) ...)`

- Apply `(lambda (b) ...)` to `#t`

- Substitute `b = #t` in

- `(if b (lambda (a b) (and a b))`  
`(lambda (a b) (or a b)))` →  
`(if #t (lambda (a b) (and a b))`  
`(lambda (a b) (or a b)))`

- *etc.*

- Lambda shielding only occurs when a function contains *nested* lambda expressions

# More Nested Lambda Examples

- Example 1:

```
(define f1
 (lambda (x y z)
 (lambda (x y) (* x y z))))
```

- (f1 3 5 4) →

```
(lambda (x y) (* x y 4))
```

- Lambda shielding only prevents substitution of x and y

- Example 2:

```
(define f2
 (lambda (x y)
 (lambda (x y z) (+ x y z))))
```

- (f2 3 5) →

```
(lambda (x y z) (+ x y z))
```

- Arguments to outer lambda are completely ignored!

# Part C: Numerical Differentiation

- Numerical differentiation is easy:
  - Some function  $f(x)$
  - Slope of  $f$  at  $x$
  - $f'(x) = ( f(x + dx) - f(x) ) / dx$
- `numerical-derivative` returns a function that implements  $f'$  using this transformation
  - Argument is a function of one argument
  - Returned function also takes one argument
  - `dx` is specified internally, e.g. with a `let` expression

# Part C: $N^{\text{th}}$ Derivative

- Example: 2<sup>nd</sup> derivative of  $f(x)$  would be:
  - (numerical-derivative  $f$ ) generates a function that calculates  $f'(x)$  using  $f(x)$
  - (numerical-derivative (numerical-derivative  $f$ )) generates another function that calculates  $f''(x)$  using  $f'(x)$
  - Need to apply numerical-derivative operation  $N$  times
    - $N$  recursive invocations of numerical-derivative
    - What should be the base case?
    - $N = 0 \rightarrow f(x)$

# Part C: Numerical Integration (1)

- Lotsa moving parts
  - Four different functions!
- make-integrator
  - Takes an “expansion function” and a  $dx$
  - Produces a function  $F2$  – the integrator itself

# Part C: Numerical Integration (2)

- Expansion function?!
  - Specifies how to integrate a function  $f$  over a specific range, e.g.  $(x - dx/2, x + dx/2)$
  - See supplemental info for more details (simple math)
- Note the arguments!
  - $f$  – some function
  - $x$  – center-point of range to integrate over
  - $dx$  – interval to integrate over

# Part C: Numerical Integration (3)

- make-integrator
  - Takes an expansion function and a step size ( $dx$ )
  - Produces a function F2 – the integrator itself
- F2 – the integrator
  - Takes a function to integrate (called F3)
  - Returns another function F4
- F3 – the function being integrated
- F4 – computes integrals of F3 over an interval
  - Takes numbers  $a$  and  $b$ ; returns a number

# Part C: Numerical Integration (4)

## ■ How does F4 work?

- Takes steps of size  $dx$ , starting at  $a$ , and ending at  $b$
- For each step, F4 uses expansion function and  $dx$  to compute integral over that step
- Keeps a running total of the integrated value
- When interval is completely traversed, return the total

## ■ Implementation:

- Recursive function
- Linear iterative or linear recursive process

# Coding Style (1)

- Indentation
  - Indent based on nesting level for more readable code
- Use line-breaks and blank lines to improve readability
  - Start defines on a new line (indent if nested)
  - Large lambda, if, cond exprs can start on a new line
  - Short expressions can be left on same line
    - (sum (lambda (x) (\* x x)) 1 6)
- Avoid really long lines (80 chars a typical max)
  - Easier to read in console window, on printouts, etc.

# Coding Style (2)

## ■ Comments!!

- Longer comments before functions; explain function's purpose and operation
- Inside functions, short comments to note details

## ■ Abstraction Hierarchy

- *Very important* in programming, in general.
- Don't cheat with cons, car, cdr, etc. 😊

## ■ Compute intermediate results

- Avoid long statements that do tons of work
- Factor out common terms, save in local variables

# Bad Coding Style

```
(define a (cons 3 5))
```

```
(define b (cons 7 8))
```

```
(define (dist p1 p2) (sqrt
 (+ (* (- (car p1) (car p2)) (- (car p1) (car p2)))
 (* (- (cdr p1) (cdr p2)) (- (cdr p1) (cdr p2))))))
```

```
(dist a b)
```

# Good Coding Style

```
;; Constructor, accessors for 2D points.
```

```
(define (make-point x y) (cons x y))
(define (get-x p) (car p))
(define (get-y p) (cdr p))
```

```
;; Compute distance between two points.
```

```
(define (distance p1 p2)
 (let ((dx (- (get-x p1) (get-x p2))) ; deltas
 (dy (- (get-y p1) (get-y p2))))
 (sqrt (+ (* dx dx) (* dy dy))))
)
)
```

```
(define a (make-point 3 5))
(define b (make-point 7 8))
(distance a b)
```