CS3:
Introduction to Symbolic Programming

Lecture 8:
Introduction to HOF

Fall 2007

Nate Titterton
nate@berkeley.edu

Schedule

8	Oct 15-19	Lecture: Higher Order Functions Lab: Introduction to HOF, lambda Reading: Simply Scheme, Ch 8, 9 (for Tue/Wed) Simply Scheme, Ch 7 (for Thur/Fri)
9	Oct 22-26	Lecture: Advanced HOF Lab: Difference between Dates, Tic Tac Toe Miniproject #3 is introduced Reading: "DbD" case study (HOF version) Simply Scheme, Ch 10
10	Oct 29 – Nov 2	Lecture: Tree Recursion, Midterm review Lab: Tree recursions Finish Miniproject #3
11	Nov 5 – 9	Lecture: *Midterm #2* Lab: Introduction to Lists

What is a procedure?

(or, a *function*).

Treating functions as things

- "define" associates a name with a value
  - The usual form associates a name with a object that is a function

```
(define (square x) (* x x))
(define (pi) 3.1415926535)
```

  - You can define other objects, though:

```
(define *pi* 3.1415926535)
(define *month-names*
   '(january february march april may
     june july august september
     october november december))
```

"Global variables"

- **Functions are "global", in that they can be used anywhere:**

```
(define (pi) 3.1415926535)
(define (circle-area radius)
        (* (pi) radius radius))
```

- **A "global" variable, similarly, can be used anywhere:**

```
(define *pi* 3.1415926535)
(define (circle-area radius)
        (*  *pi*  radius radius))
```

Are these the same?

Consider two forms of “month-name”:

```
(define (month-name1 date)
    (first date))
```

```
(define month-name2 first)
```

Procedures can be taken as arguments…

```
(define (math-function? func)
  (or (equal? func +)
      (equal? func -)
      (equal? func *)
      (equal? func /)))
```

…and procedures can be returned from procedures

```
(define (choose-func name)
   (cond ((equal? name 'plus) +)
         ((equal? name 'minus) -)
         ((equal? name 'divide) /)
         (else 'sorry)))


(define (make-add-to number)
   (lambda (x) (+ number x)))

(define joe (make-add-to 5))
```

Higher order function (HOFs)

- **A HOF is a function that takes a function as an argument.**

```
(define (do-math f arg1 arg2)
    (if (and (equal? arg2 0)
             (equal? f /))
        '(uh oh - divide by zero)
        (f arg1 arg2)))
```

The three we will focus on

- **There are three main ones that work with words and sentences:**

`every`	do something to each element
`keep`	return only certain elements
`accumulate`	combine the elements

Patterns for simple recursions

- **Most recursive functions that operate on a sentence fall into:**

Mapping:	`square-all`	`<- every`
Counting:	`count-vowels`, `count-evens`	
Finding:	`member`, `first-even`	
Filtering:	`keep-evens`	`<- keep`
Testing:	`all-even?`	
Combining:	`sum-evens`	`<- accumulate`

defining variables, `let`, and `lambda`

Three ways to define a variable

- **In a procedure call (e.g., the variable `proc`):**

```
(define (doit proc value)
     ;; proc is a procedure here…
     (proc value))
```

3. **As a global variable**

```
(define *alphabet* '(a b c d e … ))
(define *month-name* '(january … ))
```

- **With `let`**

Using `let` to define temporary variables

- **`let` lets you define variables within a procedure:**

```
(define (scramble-523 wd)
   (let ((second (first (bf wd)))
         (third (first (bf (bf wd))))
         (fifth (item 5 wd))
         )
     (word fifth second third) ) )


(scramble-523 'meaty) ➔ yea
```

Using `let` to define temporary variables

- **Using `let` can make code more readable. Consider (same functionality as before):**

```
(define (scramble-523 wd)
   (word          (first (bf wd))
                 (first (bf (bf wd)))
                 (item 5 wd)
          )
                                    )

(scramble-523 'meaty) ➔ yea
```

Any differences?

```
(define pi 3.14159265)
(define ( alpha beta pi zeta)
    … lots of code here …
    (* pi radius)
    … more code here …
```

YES!

```
(define ( alpha beta pi zeta)
  (let ((pi 3.14159265)) )
    … lots of code here …
    (* pi radius)
    … more code here …
```

Anonymous functions:

using `lambda`

the `lambda` form

- **"`lambda`" is a special form that returns a function:**

```
(lambda (arg1 arg2 …)
   statements
      )
```

```
(lambda          (x)          (*     x     x))
```

⇨ ⇨ ⇨ ⇨ ⇨

a procedure that takes one argument and multiplies it by itself

Using `lambda` with `define`

- **These are the same:**

```
(define (square x)
   (* x x))


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

Using `lambda` with `define`

- **These are VERY DIFFERENT:**

```
(define (adder-1 y)
   (lambda (x) (+ x 1)))
```

```
(define adder-2
   (lambda (x) (+ x 1)))
```

Can a `lambda`-defined function be recursive?

```
(lambda (sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (???? (bf sent)))))
```

When do you NEED lambda?

1. **When you need the context (inside a two-parameter procedure)**

```
(add-suffix '-is-great '(haddad sam mary))
    ➔ (haddad-is-great sam-is-great
       mary-is-great)
```

- **When you need to make a function on the fly**