
CS3:

Introduction to Symbolic Programming

Lecture 8:

Introduction to Higher Order Functions

Spring 2008

Nate Titterton
nate@berkeley.edu

Schedule

8	Mar 10-14	Lecture: Higher Order Functions Lab: (Tu/W) Introduction to HOF (Th/F) Using “lambda” Reading: Simply Scheme, Ch 7-9 (7 and 8 [except repeated] for Tue/Wed) (9 for Thur/Fri) Note: MP#2 due Tue/Wed
9	Mar 17-21	Lecture: Advanced HOFs Lab: Advanced HOF, tic-tac-toe Reading: "DbD" HOF version Simply Scheme, Chap 10
10	Mar 24-28	<i>Spring Break!</i>
11	Mar 31 – Apr	Tree Recursion Mini-Project #3
12	Apr 7-11	<i>Midterm #2</i>

**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))
```

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)))
```

```
STk> ((choose-func 'plus) 3 5)
```

```
8
```

```
STk> ((choose-func 'minus) 3 5)
```

```
-2
```

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`

The tricky part... (maybe)

- **Writing HOFs (even these three) is easy.**
 - You'll do that in lab Tue/Wed
- **Predicting what HOFs return is easy.**
- **Using HOFs with existing functions (like `square` to emulate `square-all`) is pretty easy.**
- **Writing procedures to be used by HOFs to solve non-trivial problems... seems to trip students up.**
 - More so if the solution involves multiple HOFs

defining variables, `let`, and `lambda`

"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))
```

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)))
```



Schedule

8	Mar 10-14	Lecture: Higher Order Functions Lab: (Tu/W) Introduction to HOF (Th/F) Using “lambda” Reading: Simply Scheme, Ch 7-9 (7 and 8 [except repeated] for Tue/Wed) (9 for Thur/Fri) Note: MP#2 due Tue/Wed
9	Mar 17-21	Lecture: Advanced HOFs Lab: Advanced HOF, tic-tac-toe Reading: "DbD" HOF version Simply Scheme, Chap 10
10	Mar 24-28	<i>Spring Break!</i>
11	Mar 31 – Apr	Tree Recursion Mini-Project #3
12	Apr 7-11	Midterm #2



Remember, in STk, when you type something, Scheme is going to (after evaluating it) print out the (return) value as best it can. A number is printed out directly. A word, or sentence, is also directly printed (but, without the quote). But what does a function look like?

Try the following from STk:

```
STk> +  
#[closure arglist=args 753de8]  
STk> (define (plus num1 num2) (+ num1 num2))  
plus  
STk> plus  
#[closure arglist=(num1 num2) 12605d8]  
STk> (define (square num) (* num num))  
square  
STk> square  
#[closure arglist=(num) 125c298]  
STk> (define (square num) (* num num))  
square  
STk> square  
#[closure arglist=(num) 125d608]
```





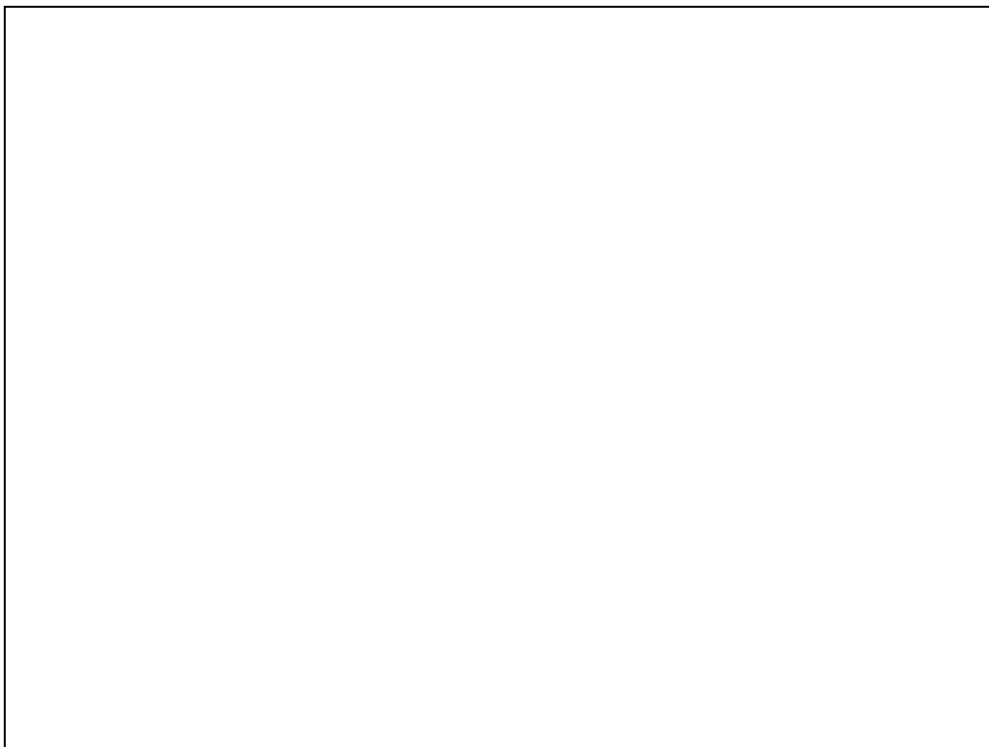

Yep, these are pretty much the same in practice.

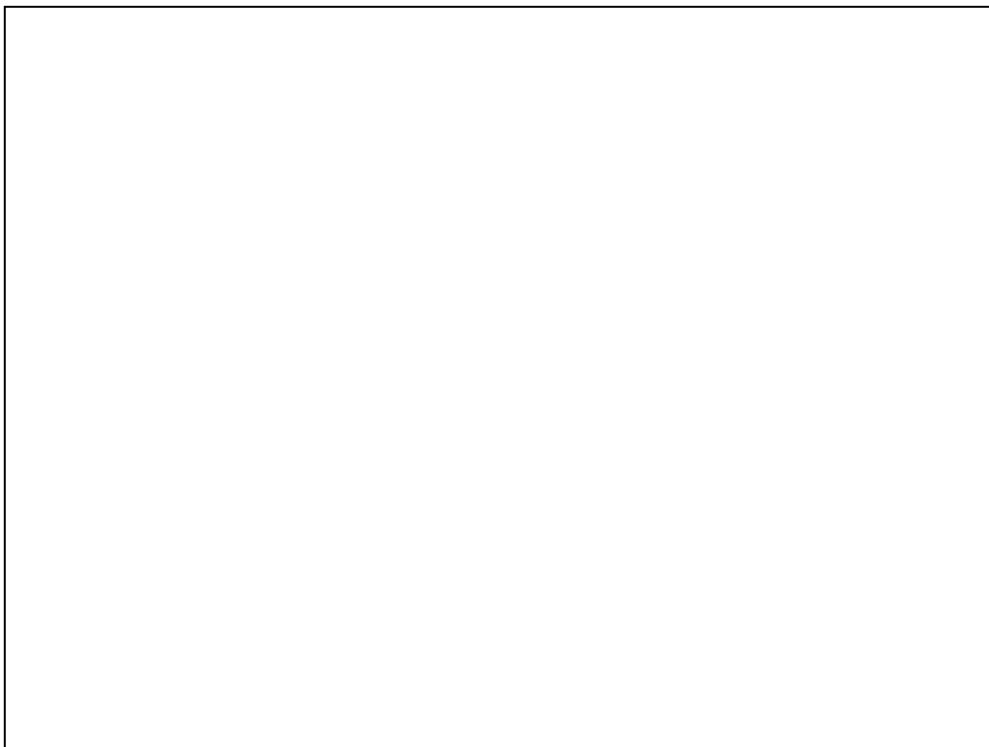
In lecture, we also showed:

```
(define (plus num1 num2)
  (+ num1 num2))
```

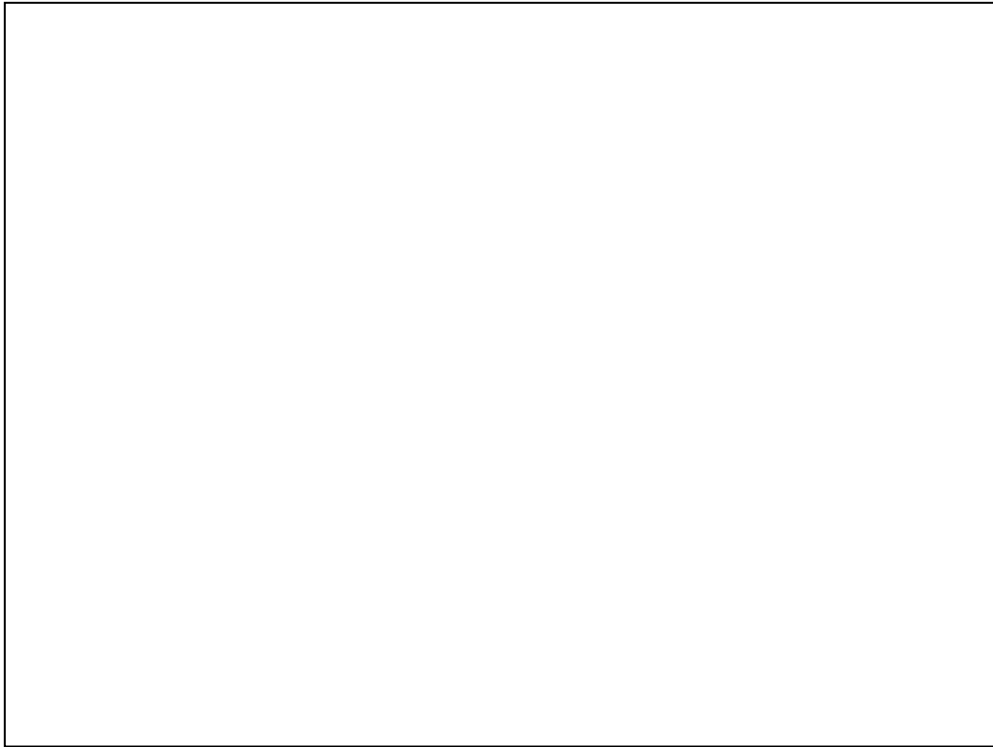
```
(define plus2 +)
```

in this case, “plus” and “plus2” are different in the number of arguments that they can take (“plus2” can take any number of numeric arguments, “plus” can only take 2).









Every takes two arguments: a function and a sentence (or word). The function takes one argument, and is called on every element of the sentence (or word)

```
(define (factorial n)
  (if (< n 1) 1 (* n (factorial (- n 1)))))

(every factorial '(1 2 3 4 5)) --> (1 2 6 24 120)
```

Keep takes two arguments: a predicate (function) and a sentence (or word). The predicate takes one argument, and is called on each element of the sentence or word.

```
(keep odd? '(1 2 3 4 5 6 7)) --> (1 3 5 7)

(define (vowel? ltr) (member? ltr '(a e i o u)))
(keep vowel? 'mississippi) --> iiii
```

Accumulate takes two parameters: a function and a sentence (sometimes a word). The function here, however, takes two arguments.

```
(accumulate + '(1 2 3 4 5)) --> 15
(accumulate word '(t hee ndisn ear)) -->
```



The tricky part... (maybe)

- **Writing HOFs (even these three) is easy.**
 - You'll do that in lab Tue/Wed
- **Predicting what HOFs return is easy.**
- **Using HOFs with existing functions (like `square` to emulate `square-all`) is pretty easy.**
- **Writing procedures to be used by HOFs to solve non-trivial problems... seems to trip students up.**
 - More so if the solution involves multiple HOFs

defining variables, `let`, and `lambda`



The asterisks are convention, not required by scheme. Generally, when you surround a global variable with asterisks, you differentiate it from other variables you might be using inside functions (which, right now, are passed as parameters). So, also by convention, don't surround parameter names with asterisks!

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))
)
```

The top form is just a shortcut, really, for the bottom form. We would get tired having to type l-a-m-b-d-a all the time, so the above form is quicker.

Using lambda with define

- These are VERY DIFFERENT:

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

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

adder1 takes a single argument and returns a procedure (that takes a single argument and returns 1 more than it)

adder2 takes a single argument and returns one more than it.