CS3:
Introduction to Symbolic Programming

Lecture 9:
More HOF
tic-tac-toe

Spring 2008

Nate Titterton
nate@berkeley.edu

Schedule

9	Mar 17-21	Lecture: Advanced HOFs Lab: Advanced HOF, tic-tac-toe Miniproject #3 introduced Reading (Tue/Wed): "DbD" HOF version Simply Scheme, Chap 10
10	Mar 24-28	*Spring Break!*
11	Mar 31 – Apr 4	Lecture: Tree Recursion, Midterm review Lab: Tree Recursion Mini-Project #3 (Due Friday at midnight)
12	Apr 7-11	*Midterm #2* Lab: Introduction to lists
13	Apr 14-18	Advanced lists…

You can work on mini-project #3 this week

	Tue/Wed	Thur/Fri
This week		Miniproject introduced, ½ lab to work on it
Next week	*Spring Break*	
Third week	Full day of tree recursion!	A few review materials introduced. Otherwise, open lab. MP#3 due Friday at midnight
MIDTERM #2…		

Tic Tac Toe

The board

```
 x  |    |
----+----+----
 o  | o  | x
----+----+----
    |    |
```

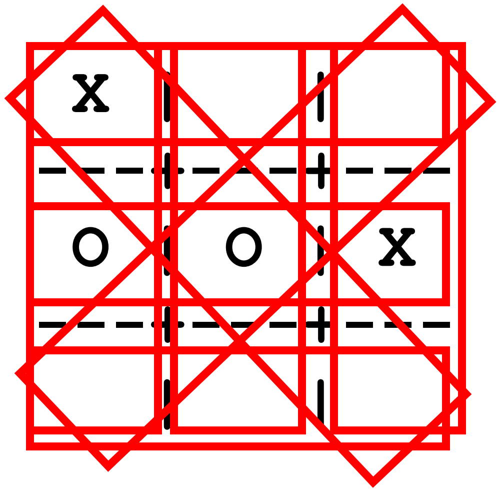
```
"x _ _"

"o o x"

"_ _ _"
```

"X _ _ O O X _ _ _"

Triples (another representation of a board)



"X _ _ O O X _ _ _"

```
(  x23  oox  789  xo7  2o8  3x9  xo9  3o7  )
```

Tic-tac-toe hints

- Read the chapter!
- You will need to be familiar with vocabulary
  - positions, triples, "forks", "pivots", and so on
- This chapter in the book comes *before* recursion.
  - You would solve things differently if you used recursion
- The code (at the end of the chapter) has no comments.

Higher-order functions: review

Higher order function (HOFs)

- A HOF is a procedure that takes a procedure as an argument.
- There are three main ones that work with words and sentences:
  - `every`
    - take a one-argument procedure that returns a word
    - do something to each element
  - `keep`
    - takes a one-argument predicate
    - return only certain elements
  - `accumulate`
    - takes a two-argument procedure
    - combine the elements

A definition of every

```
(define (my-every proc ws)
  (if (empty? ws)
    '()
    (se (proc (first ws))
        (my-every (bf ws))
        )))
```

- **HOFs do a lot of work for you:**
  - Checking the conditional
  - Returning the proper base case
  - Combing the various recursive steps
  - Invoking themselves recursively on the smaller problem

Accumulate: right to left!

- The *direction* matters: right to left
  - `(accumulate / '(4 2 2))` does not equal `1`, but `4`.
- Think about expanding an accumulate

```
(accumulate + '(1 2 3 4))
  ➔ (+ 1 (+ 2 (+ 3 4)))

(accumulate / '(4 2 2))
  ➔ (/ 4 (/ 2 2))
```

Consider how accumulate is written…

```
(define (my-accum1 accum-proc sent)
  (if (= (count sent) 1)   ;;last element

    (first sent)

    (accum-proc
       (first sent)
       (my-accum1 accum-proc (bf sent)) ) ) )
```

Accumulate: returning sentences

- `accumulate` can return a sentence…

```
(accumulate ?? '(a b c d))
```

$\rightarrow$ `(ab bc cd)`

  - the *first* time accumulate is run, it reads the last two words of the input sentence
  - in *later* calls, it uses the return value of its procedure (which is a sentence) as one of its arguments

Any questions from Tue/Wed last week?

- You wrote and played with `every`, `keep`, and `accumulate`
- You used them in combination:

```
(remove-adj-dupls 'mississippi)
 ➔ misisipi

(gpa '(A A F C B))
    ➔ 2.6   (average of 4, 4, 0, 2, 3)

(gpa-with-p/np '(A A F NP P C B))
    ➔ 2.6   (average of 4, 4, 0, 2, 3)

(true-for-all? even? '(2 4 6 8))
    ➔ #t
```

Which HOFs would you use? (1/2)

1) **capitalize-proper-names**

```
(c-p-n '(mr. smith goes to washington))
     ➔ (mr. Smith goes to Washington)
```

2) **count-if**

```
(count-if odd? '(1 2 3 4 5)) ➔ 3
```

3) **longest-word**

```
(longest-word '(I had fun on spring
   break)) ➔ spring
```

4) **count-vowels-in-each**

```
(c-e-l '(I have forgotten everything))
   ➔ (1 2 3 3)
```

Which HOFs would you use? (2/2)

5) **squares-greater-than-100**

```
(s-g-t-100 '(2 9 13 16 9 45))
   ➔ (169 256 2025)
```

6) **root of the sum-of-squares**

```
(sos '(1 2 3 4 5 6 7))
   ➔ (sqrt (+ (* 1 1) (* 2 2) …)
   ➔ 30
```

7) **successive-concatenation**

```
(sc '(a b c d e))
   ➔ (a ab abc abcd abcde)
```

Any questions from Thur/Fri last week?

- You wrote and played with `lambda` and `let`

Three ways to define a variable

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

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

2. **As a global variable**

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

3. **With `let`**

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
```

Use lambda anywhere you need a function

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


(every (lambda (x) (* x x))
       '(1 2 3))
  ➔ (1 4 9)


((lambda (x) (* x x))  3)
  ➔ 9
```

You *need* lambda when…

…you need a procedure to make reference to more values than you can pass it.

For instance, when a procedure for use in an `every` needs two parameters

```
(prepend-every 'sir- '(sam mary loin))
    ➔ (sir-sam sir-mary sir-loin)
```

Write `prepend-every`

Write `appearances`

`make-bookends` (a *small* problem)

- Write `make-bookends`, which is used this way:

```
((make-bookends 'o) 'hi) → ohio

((make-bookends 'to) 'ron) → toronto

(define tom-proc (make-bookends 'tom))
(tom-proc "") → tomtom
```

Problems

C’mon on down

"The Price is Right" was a game show which started each round in the same way: several guests would guess at the price of an item, trying to get as close as possible to the real price without going over. The person who was closest without going over would win.

In this question, you will work with a procedure PIR-winner which takes a sentence of guesses and a real price, and returns the guess that is closest without being greater than the real price:

```
(PIR-winner '(134 245 199 300 160) '200)  → 199
(PIR-winner '(19 35 56 44 22) '35)        → 35
```

(You can assume that there is always at least one guest whose guess is below or equal to the real price, and that no two guesses will be the same).

Write as a HOF.