CS3:
Introduction to Symbolic Programming

Lecture 9:
Higher Order Procedures

Spring 2006

Nate Titterton
nate@berkeley.edu

Schedule

8	Mar 6-10	Lecture: Finishing recursion Lab: Miniproject #2: Number names
9	Mar 13-17	Introduction to Higher Order Procedures Reading: SS 7-9; "DbD" part III
10	Mar 20-24	More HOF, Tic-Tac-Toe, Tree Recursion Reading: SS 10, 15; "Change Making" case study
11	Mar 27-31	*(Spring Break)*
12	Apr 3-7	Lecture: Review Lab: Miniproject #3
13	Apr 10-14	Lecture: MIDTERM #2 Lab: Start on "Lists"

Announcements

- **Mid-semester survey this week (Thurs/Fri)**
  - You need to do this
- **Reading this week:**
  - Simply Scheme Chapters 7-9
  - Difference between dates, part III

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

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

```
(define *pi* 3.1415926535)
(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)
```

Why have procedures as objects?

Other programming languages don’t (often)

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 add-to-5 (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

Using every...

```
(define (square-all sent)
   (if (empty? sent)
       '()
       (se (square (first sent))
           (square-all (bf sent))
       ))

(square-all '(1 2 3 4 5))



(every square '(1 2 3 4 5))
```

Write "my-every"

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

Write "my-keep"

```
(my-keep odd? '(1 2 3 4 5))
➔ (1 3 5)
```

lambda

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

```
(lambda (param1 param2 …)
   statement1
   statement2
      )

(lambda (x) (* x x)) ➔ [a function]
(every (lambda (x) (* x x)) '(1 2 3 4))
    ➔ (1 4 9 16)
```

Using `lambda` with `define`

- **Is there a difference between:**

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

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

How about between…

```
(define (special? wd)
   (member? wd (member wd '(a b c x y z))))

(define (big-proc ...)
  ... lots of code ...
  (keep special? a-sentence)
  ... more code ... )
```

```
(define (big-proc ...)
  ... lots of code ...
  (keep (lambda (wd)
          (member wd '(a b c x y z)))
        a-sentence)
  ... more code ... )
```