CS107 Handout 61
Autumn 2000 December 7, 2000

Lambda

Written by Todd Feldman, Jerry Cain, and Andy Gray

Lambda

Writing a lambda statement basically defines a function on the fly, but without giving it a name. Sometimes the concept of such an "anonymous function" bothers people who are used to programming in C or Pascal, since every function has to have a name from somewhere.

Let's write the `repeat` function: it takes a number and a one-argument function, and returns a new function that acts like the original function argument, except that it causes the function to be repeated on its output a number of times. It recursively creates a function which is a general case of these two less elegant functions:

```
(defun do-twice (function)
    #'(lambda (argument) (funcall function
   (funcall function argument))))

(defun do-thrice (function)
   #'(lambda (argument) (funcall function
   (funcall function
(funcall function argument))))

;; repeat:
;;   Takes a one-argument function and an (integer) number as arguments.
;;   Returns a new function which is the composition of the function with
;;   itself <number> times.
;;
(defun repeat (fxn num)
   (if (= num 0)
       #'(lambda (arg) arg)            ;; Base case: Identity function
       #'(lambda (arg) (funcall fxn (funcall (repeat fxn (- num 1)) arg)))
   ;; Recursive case: Makes a new function that calls FXN on the function
   ;; returned from calling REPEAT with NUM subtracted by 1.
   ))
```

Perhaps it's a little hard to figure out, but take the time— it's an excellent, concise example of the power of recursion and first-order functions. `(repeat #'double 2)` is the same as `(do-twice #'double)`, and has the advantage of not forcing you to write all sorts of versions of essentially the same thing.

Example:
An obvious use for the repeat function would be for exponentiation:

```
> (* 8 8)
64
> (defun times-num (num)  #'(lambda (argument) (* num argument)))
TIMES-NUM
> (defun square (number) (funcall (do-twice (times-num number)) 1))
SQUARE
> (square 8)
64
> (defun another-square (number) (funcall (repeat (times-num number) 2) 1))
ANOTHER-SQUARE
> (another-square 8)
64
> (defun power (number exp)  (funcall (repeat (times-num number) exp) 1))
POWER
> (power 2 10)
1024
```

Selectors: Good Design and Good Readability

Finally, a word about selectors. Your programs will run fine without them, but they help code maintenance, debugging, and readability in an incredible way.

Suppose we have a database full of grades, in which a student's grade file is laid out in a list, like this:

(*<name>* (*<hw1> <hw2> <hw3> <hw4>*) *<mt1> <mt2> <final> <XC> <participation>*)

You could write a function to compute average homework score like this, but it's obvious that this horrid little beast is going to get you a Minus for Readability.

```
(defun homework-average (gradentry)
   ( /
      ( + (caadr gradentry) (cadadr gradentry) (caddr (cadr gradentry))
          (cadddr (cadr gradentry)))
      4
   ))
```

But in addition to lack of clarity, debugging it can be hard! What if you get one of the cXr's wrong? The debugger tells you that you've passed an incorrect argument to `car` or `cdr` in the function `homework-average`, which could be anywhere in this function. And what if the original programmer decided he/she wanted to change the layout of the grade entry list? That would mean that the `homework-average` function would need to be rewritten. Also, chances are that homework-average is not the only functions written in this style! Change the list layout, and all the functions need to be rewritten. Yuck! However, if you add a few selectors, you get a lot of maintainability in return.

```
(defun homework1 (grade-entry) (caadr grade-entry))
(defun homework2 (grade-entry) (cadadr grade-entry))
```

```
(defun homework3 (grade-entry) (caddr (cadr grade-entry)))
(defun homework4 (grade-entry) (cadddr (cadr grade-entry)))

(defun homework-average (grade-entry)
   (let (( hw1 (homework1 grade-entry) )
         ( hw2 (homework2 grade-entry) )
         ( hw3 (homework3 grade-entry) )
         ( hw4 (homework4 grade-entry) ))
      (/ (+ hw1 hw2 hw3 hw4) 4) ))
```

(Of course, this function could have been better if written recursively, so that you can select out the whole list of homeworks in the grade file and allow the instructor to vary the number of assignments, but I'm just trying to prove a point here.) If the layout of the grade entry list happened to change, I could leave the homework-average function alone, and change my definitions of the `homework1-4` selectors instead. Imageine that you have 100 functions like `homework-average`, which extract one or more homework scores. Without the selectors, you'd have to update 100 functions. But with selectors, you only have to update the homework and other selectors - that's only 9 functions as opposed to 100, and you're less likely to make a mistake somewhere. Besides, this code is much more readable.

More Hints on Improving LISP Design & Readability (or, how to make your TA happy)

· Break long functions into smaller helpers, each of which does one or two tasks. This not only makes your code readable, it alos lets you test the helpers quickly and helps you pinpoint problems easily.

· Use `mapcar` (instead of recursive function) to apply some action on each element of a list. Also, using helpers to tailor data into a form that `mapcar` can operate on is more preferable to creating a new recursive function that performs tests on elements, puts elements back into a list, and recurses.

· Comments inlined in a function could actually make it less readable. Since our functions are small and single-minded, having good comments before the function usually work nicely.

· `If #'(lambda …)` is getting too long, make it into a named function.

· Use `let` or `let*` sparingly. These are mostly appropriate when you need to avoid re-evaluating an expensive expression. They may also be used to aid readability by giving names to intermediate steps of the computation. But be wary of allowing yourself to fall back into the procedural paradigm by writing your program as a series of let bindings!

Problem 1

a. Let's write a handy function `accum`, which takes three arguments: a function of two arguments, called the "acumulator", a list, and a "default value". Basically, `accum` is a generic recursion function, applying the accumulator to the first element of the list, and the result of calling `accum` on the rest, or if the rest is empty, on the default value.

```
> (accum #'(lambda (x) x) '() 'this-is-the-default-value)
this-is-the-default-value
> (accum #'+ '(1 2 3) 0)      ;; sum-list operation using accum
6
> (accum #'* '(2 3 5) 1))     ;; product-list operation using accum
30
```

b. Code reuse exercise: Redefine `forall` from the previous problem using `accum`. Now, can you think of a way of redefining `append` using `accum`? What about `mapcar`?

<u>Solution</u>

```
;;;;;;;;;;;;
;;; Function: accum
;;;;;;;;;;;;
;;; accum takes three parameters, a function (of two arguments), a list, and
;;; a "default value".  It returns the result of "accumulating" the list by
;;; applying the given function on the first element of the list and the
;;; result of accum on the rest of the list.
;;;;;;;;;;;;

(defun accum (operator list default-value)
  (if (null list) default-value
      (funcall operator (car list)
                        (accum operator (cdr list) default-value))))

;;;;;;;;;;;
;;; Redefining forall, append and mapcar using accum
;;;;;;;;;;;

(defun forall (pred list)
  (accum #'(lambda (elmt lst)
             (and (funcall pred elmt) lst)) list T))

(defun append (list1 list2)
  (accum #'cons list1 list))

(defun mapcar (func list)
  (accum #'(lambda (elmt lst) (cons (funcall func elmt) lst)) list '()))
```

Problem 2

Write a function `increasing-sum` that takes a list of non-empty integer lists, such as `'((2 4 3) (6 2) (5 6 7 3))`. The function should return a list where the sublists are arranged in increasing order by their sum.

```
> (increasing-sum '((2 4 3) (6 2) (5 6 7 3)))
((6 2) (2 4 3) (5 6 7 3))
>
```

Next, write a function `increasing-max` that takes a list of non-empty integer lists, and returns a new list where the sublists are arranged in increasing order by their maximum element.

```
> (increasing-max '((10) (4 0 1) (4 3 2 8)))
((4 0 1) (4 3 2 8) (10))
> (increasing-max '((9) (8) (7) (6) (5) (4) (3) (2) (1)))
((1) (2) (3) (4) (5) (6) (7) (8) (9))
> (increasing-max '((1 10) (2 9) (3 8) (4 7) (5 6) (6 5) (7 4) (8 3) (9 2)
(10 1)))
((5 6) (6 5) (4 7) (7 4) (3 8) (8 3) (2 9) (9 2) (1 10) (10 1))
>
```

Solution

```
;;;;;;;;;;;;;
;;; Function: sum-compare
;;;;;;;;;;;;;
;;; sum-compare takes in two lists of integers as parameters, list-one
;;; and list-two, and returns TRUE if and only if the sum of the integers
;;; in list-one is less than the sum of the integers in list-two
;;;;;;;;;;;;;

(defun sum-compare (list-one list-two)
  (< (apply #'+ list-one) (apply #'+ list-two)))

;;;;;;;;;;;;;
;;; Function: increasing-sum
;;;;;;;;;;;;;
;;; increasing-sum takes in a list of non-empty integer lists, and
;;; returns a list where the integer lists are arranged in increasing order
;;; by their sum.
;;;;;;;;;;;;;

(defun increasing-sum (list-of-lists)
  (sort list-of-lists #'sum-compare))
```

```
;;;;;;;;;;;;;
;;; Function: max-compare
;;;;;;;;;;;;;
;;; this function compares two incoming lists of integers,
;;; list-one and list-two, and returns TRUE if and only
;;; if the maximum element of list-one is smaller than the
;;; maximum element of list-two
;;;;;;;;;;;;;

(defun max-compare (list-one list-two)
  (< (apply #'max list-one) (apply #'max list-two)))

;;;;;;;;;;;;;;
;;; Function: increasing-max
;;;;;;;;;;;;;;
;;; increasing-max takes a list of non-empty integer lists, and
;;; returns a list where the integer lists are arranged in increasing order
;;; by their maximum element
;;;;;;;;;;;;;;

(defun increasing-max (list-of-lists)
  (sort list-of-lists #'max-compare))

;;;;;;;;;;;;;;
;;; Function: cool-increasing-max
;;;;;;;;;;;;;;
;;; Note that we could have spared ourselves the chore of defining
;;; a separate max-compare function by passing in the equivalent
;;; lambda expression as the second argument to sort.
;;;;;;;;;;;;;;

(defun cool-increasing-max (list)
  (sort list #'(lambda (list-one list-two)
                 (< (apply #'max list-one) (apply #'max list-two)))))
```

Problem 3

Write a function called swap that exchanges any two elements of a list. swap takes three parameters: a list and two integers (representing the positions of the elements to be exchanged.) Assume that elements of a list are indexed from 0.

```
> (swap '(HOMER MARGE BART LISA MAGGIE) 1 3)
(HOMER LISA BART MARGE MAGGIE)
> (swap '(MIKE CAROL GREG MARCIA PETER JAN BOBBY CINDY ALICE) 1 8)
(MIKE ALICE GREG MARCIA PETER JAN BOBBY CINDY CAROL)
>
```

<u>Solution</u>

```
;;;;;;;;;;;;
;;; Function: delete-first-n
;;;;;;;;;;;;
;;; delete-first-n takes two parameters -- specifically, it takes a list
;;; and an integer n -- and returns a copy of the incoming list with
;;; the first n elements removed
;;;;;;;;;;;;

(defun delete-first-n (list n)
  (cond ((equal n 0) list)
        ((null list) (error "Illegal operation on an empty list"))
        (T (delete-first-n (cdr list) (1- n)))))


;;;;;;;;;;;;
;;; Function: sub-list-helper
;;;;;;;;;;;;
;;; sub-list-helper takes as parameters a list and two integers, front-chop
;;; and backchop. sub-list-helper removes the first front-chop entries and
;;; truncates the last back-chop and the first front-chop entries and
returns
;;; the new list.
;;;;;;;;;;;;

(defun sub-list-helper (list front-chop back-chop)
  (delete-first-n (reverse (delete-first-n (reverse list) back-chop))
front-chop))


;;;;;;;;;;;;
;;; Function: sub-list
;;;;;;;;;;;;
;;; sub-list takes three arguments: a list, and two list indices.
;;; sub-list will remove the elements up to but not including
;;; the first index, and will truncate those beyond the second index.
;;; elements of the list are indexed from 0
;;;;;;;;;

(defun sub-list (list start final)
  (if (< final start) '()
      (sub-list-helper list start (- (length list) final 1))))
```

```
;;;;;;;;;;;;
;;; Function: swap
;;;;;;;;;;;;
;;; swap takes three parameters: a list and two integers (representing the
;;; positions of the elements to be exchanged by the swap.)  Assume that elements
;;; are indexed from 0.
;;;;;;;;;;;;

(defun swap (list first second)
  (if (equal first second) list
      (let* ((lower (min first second))
             (higher (max first second))
             (back (sub-list list (1+ higher) (1- (length list))))
             (middle (sub-list list (1+ lower) (1- higher)))
             (front (sub-list list 0 (1- lower))))
         (append front
                  (list (nth higher list))
                  middle
                  (list (nth lower list))
                  back))))
```