higher order functions

---

brief

Higher Order functions are functions which either (i) take other functions as arguments and/or (ii) return functions as results.

These notes consider a couple of issues.

standard Lisp functions which take function arguments

Many of the standard Lisp functions can take function arguments to change their behaviour. We have (probably) already seen some of these.

In the following example we use the function #'get-num as the :key argument for find...

```
(defparameter *english* '((one 1)(two 2)(three 3)(four 4)(five 5)))

(defparameter *hindi* '((ek 1)(do 2)(tin 3)(char 4)(panch 5)))

(defun get-word (pair) (first pair))
(defun get-num (pair) (second pair))

(defun translate (word from to)
  (get-word (find (get-num (find word from :key #'get-word))
                  to :key #'get-num)))


cg-user(6): (translate 'three *english* *hindi*)
tin

cg-user(7): (translate 'char *hindi* *english*)
four
```

using functions to manufacture other functions

In the next example we write a function to make a function & then use this as an argument to remove-if...

```
(defun make-memberp (members)
  #'(lambda (x) (member x members)))


cg-user(14): (setf noun? (make-memberp '(cat dog bat frog)))
#<Closure (:internal make-memberp 0) @ #x21066f5a>

cg-user(15): (setf verb? (make-memberp '(eat meet greet)))
#<Closure (:internal make-memberp 0) @ #x210746d2>

cg-user(16): (funcall noun? 'bat)
(bat frog)

cg-user(17): (funcall verb? 'bat)
nil

cg-user(18): (funcall verb? 'eat)
(eat meet greet)

cg-user(20): (remove-if noun? '(cat greet dog meet bat eat frog))
(greet meet eat)
```

making functions out of functions

This example writes a function to apply 2 others. It forms a conjunction of 2 functions...

```
(defun fn-conjunctor (f1 f2)
  ;; make a conjunction of 2 functions
  #'(lambda (x) (funcall f2 (funcall f1 x))))


(defun 100+ (x)
  ;; a simple function to increment by 100
  (+ x 100))


cg-user(21): (setf fx1 (fn-conjunctor #'1+ #'100+))
#<Closure (:internal fn-conjunctor 0) @ #x21117fda>

cg-user(22): (funcall fx1 23)
124

cg-user(27): (setf fx2 (fn-conjunctor #'second #'100+))
#<Closure (:internal fn-conjunctor 0) @ #x211594b2>

cg-user(28): (mapcar fx2 *english*)
 (101 102 103 104 105)
```

local variables & closures

an example of a function referencing a variable from its local closure

```
(defun make-counter (init-val)
  (let ((counter init-val))
    #'(lambda () (setf counter (1+ counter)))
    ))


cg-user(8): (setf c1 (make-counter 10))
#<Closure (:internal make-counter 0) @ #x21181572>

cg-user(9): (setf c2 (make-counter 100))
#<Closure (:internal make-counter 0) @ #x21186c1a>

cg-user(10): (funcall c1)
11

cg-user(11): (funcall c2)
101

cg-user(12): (funcall c2)
102

cg-user(13): (funcall c1)
12
```

using closures to make data streams

In this example we use a closure to wrap a list & then use funcall to step through the list. The example is contrived (it would be easier just to use pop) but it demonstrates a point.

```
(defun make-data-stream (lis)
  #'(lambda () (pop lis)))

;; set up a data stream generator for a couple of lists
cg-user(15): (setf fruit (make-data-stream '(mango melon pawpaw peach)))
#<Closure (:internal make-data-stream 0) @ #x10e22952>

cg-user(16): (setf fish (make-data-stream '(cod kipper carp sardine)))
#<Closure (:internal make-data-stream 0) @ #x10e2af42>

>> (funcall fruit)  →  mango
>> (funcall fruit)  →  melon
>> (funcall fish)  →  cod
>> (funcall fish)  →  kipper
>> (funcall fish)  →  carp
>> (funcall fish)  →  sardine
```

combining functions

A function to run two other functions...

```
(defun fn-conj2 (f1 f2)
  #'(lambda (x) (funcall f2 (funcall f1 x))))

cg-user(2): (setf foo (fn-conj2 #'1+ #'evenp))
#<Closure (:internal fn-conj2 0) @ #x10eee3aa>

cg-user(4): (funcall foo 7)
t

;; a throwaway fn to help testing...
(defun 10* (n) (* n 10))

cg-user(71): (setf bar (fn-conj2 #'10* #'sqrt))
#<Closure (:internal fn-conj2 0) @ #x10f516da>

cg-user(72): (funcall bar 20)
14.142136
```

Now a general purpose function to apply a list of functions...

```
(defun fn-conj-many (fn-list)
  (reduce #'fn-conj2 fn-list))


cg-user(74): (setf boo (fn-conj-many (list #'first #'1+ #'10* #'sqrt)))
#<Closure (:internal fn-conj2 0) @ #x10de5afa>
cg-user(75): (funcall boo '(1 banana))
4.472136
```

a simple *Objects* system(?)

In this example we wrap values but export lambda forms (methods) to access & modify those variables. Then we build a *helper* function to access & apply these lambda forms.

The example uses a simple structure to hold a student name & a set of marks...

```
;; specify data & "methods" then export the methods
(defun make-student (name)
  (let* (marks
         (methods
          `((get-name  . ,#'(lambda() name))
            (add-mark  . ,#'(lambda(m) (push m marks)))
            (get-marks . ,#'(lambda() marks))
            (ave-mark  . ,#'(lambda() (/ (reduce #'+ marks)
                                         (length marks))))
           ))
         )
    methods
    ))

;; a function to access & apply methods
(defun => (obj method &rest args)
  (apply (-> obj method) args))


cg-user(27): (setf ralf (make-student 'ralf))
((get-name . #<Closure (:internal make-student 0) @ #x10dc857a>)
 (add-mark . #<Closure (:internal make-student 1) @ #x10dc8592>)
 (get-marks . #<Closure (:internal make-student 2) @ #x10dc85aa>)
 (ave-mark . #<Closure (:internal make-student 3) @ #x10dc85c2>))

>> (=> ralf 'get-name)  →  ralf
>> (=> ralf 'add-mark 56)  →  (56)
>> (=> ralf 'add-mark 64)  →  (64 56)

cg-user(32): (setf sue (make-student 'susan))
((get-name . #<Closure (:internal make-student 0) @ #x10e321b2>)
 (add-mark . #<Closure (:internal make-student 1) @ #x10e321ca>)
 (get-marks . #<Closure (:internal make-student 2) @ #x10e321e2>)
 (ave-mark . #<Closure (:internal make-student 3) @ #x10e321fa>))

>> (=> sue 'add-mark 55)  →  (55)
>> (=> sue 'add-mark 73)  →  (73 55)
>> (=> sue 'add-mark 69)  →  (69 73 55)
>> (=> ralf 'get-marks)  →  (64 56)
>> (=> sue 'get-marks)  →  (69 73 55)
>> (=> ralf 'ave-mark)  →  60
>> (=> sue 'ave-mark)  →  197/3
```