Functional Programming 2

jens.jensen@stfc.ac.uk
0000-0003-4714-184X

March 13, 2022

Outline of Talk 2

- Previous talk (talk 1):
  - Introibo
  - Pure Functional Programming Principles
- This talk (and probably the next):
  - Mapping
  - Labels and naming
  - Lists
- Advanced(ish) Topics
- Impure Functional? Side Effects
- Category Theory
- Categories and Functions
- Categories and Computation

Still written in the author's spare time!

Very much a personal perspective, and not following any particular textbook. Using *meditations* and *exercises* – solutions to all exercises given during the talks.

Summary of Talk 1

- Basic Principles
  - Variables are immutable
  - Functions have no side effect
    - Side effects permitted on r-values (often preferred for efficiency)
    - In this talk we look at functions with side effects
  - "Divide and conquer" approach to problems

Summary of Talk 1

```
(defun fact (k)
  "Calculate the factorial of a number"
  (unless (and (numberp k) (integerp k) (>= k 0))
    (error "Unable to take factorial of %s" k))
  (fact-1 k))
```

`fact` is the entry point; it delegates to a helper function which is guaranteed to be called with a non-negative integer. This means that (in principle) checks can be turned off for the helper, and it can be optimised to run faster:

```
(defun fact-1 (k)
  (if (zerop k) 1 (* k (fact-1 (1- k)))))
```

More Mapping

Meditation exercise: why do mapping *et al.* not work with macros and special forms?

```
(apply #'and '(nil t t))
```

raises an error, though these will work in EL, but not in CL:

```
(funcall #'if t 'yes 'no)
yes
(funcall #'and t t nil)
nil
```

Example – Sorting Months

Where does August come before July, April before January?

```
(defvar +dates+ '((10 . "Aug") (2 . "Dec") (17 . "Mar") (30
```

Let's start with a function to calculate the month number:

```
(defun month-number (m)
  (1+ (floor
       (search m
         "JanFebMarAprMayJunJulAugSepOctNovDec")
       3)))
month-number
(month-number "Jan")
1
(month-number "Dec")
12
```

This function is *correct* in the sense of producing the right output given the right input, but it does have some flaws and inefficiencies (exercise) – we shall return to these later.

Example – Sorting Months

Now the problem is simple: first we add the month number...

```
(mapcar (lambda (d) (cons (month-number (cdr d)) d)) +dates-
((8 10 . "Aug") (12 2 . "Dec") (3 17 . "Mar") (4 30 . "Apr")
```

then we sort those

```
(mapcar #'cdr
 (sort
  (mapcar (lambda (d) (cons (month-number (cdr d)) d)) +date
  (lambda (d1 d2) (or (< (first d1) (first d2))
                      (and (= (first d1) (first d2))
                           (< (second d1) (second d2)))))))
((4 . "Jan") (17 . "Mar") (30 . "Apr") (2 . "Aug") (10 . "Au
```

This is what Perl calls the "Schwarzian Transform" – add the order value to the entries, sort on the order value, and then strip it off again at the end.

Example – Sorting Months

Of course with this being Lisp, we can do better:

```
(sort (copy-seq +dates+)
  (lambda (d1 d2)
    (let ((m1 (month-number (cdr d1)))
          (m2 (month-number (cdr d2))))
      (or (< m1 m2) (and (= m1 m2) (< (car d1) (car d2)))))))
((4 . "Jan") (17 . "Mar") (30 . "Apr") (2 . "Aug") (10 . "Au
```

Meditation: why do we need `copy-seq` here when we didn't need it before? Is this solution better than the Schwarzian transform?

In CL, the `:key` parameter is used to extract the field to be sorted on, but it can also be used to calculate the order:

```
(sort (copy-seq +dates+) #'<
  :key (lambda (d) (+ (car d) (* 40 (month-number (cdr d))))))
```

Example – Magic Mapping with Apply

From the author's AoC21, Day 9, in CL rather than EL:

```
(defun transpose (rows)
  "Transpose rows and columns, list of lists version"
  (apply #'mapcar #'list rows))
```

This magic works because `apply` accepts the extra argument (`#'list`) to pass to `mapcar`:

```
(transpose '((1 2 3) (4 5 6)))
((1 4) (2 5) (3 6))
```

Advanced Maps

Meditate on the advanced mapping functions: this CL example is also from AoC21 Day 9 (uses `incf` to get map-indexed):

```
(defun row-troughs (row)
  "Return the locations of the \"troughs\" in a row of
   numbers, local minima"
  ;; General case: haven't thought too much about it
  (let* ((ext-row (cons most-positive-fixnum
                        (append row
                                (cons most-positive-fixnum nil))))
         (idx 0))
    (mapcan (lambda (a b c)
              (prog1 (if (and (> a b) (< b c))
                         (list idx) nil) (incf idx)))
            ext-row (cdr ext-row) (cddr ext-row))))
```

Example Koan - fizzbuzz 1

```
(defun buzz (num) (if (zerop (mod num 5)) (list 'buzz) nil))
(defun fizz (num)
  (funcall
   (if (zerop (mod num 3))
       (lambda (x) (cons 'fizz x)) #'identity)
   (buzz num)))
(fizz 2)
nil
(fizz 6)
(fizz)
(fizz 10)
(buzz)
(fizz 30)
(fizz buzz)
```

Meditation: why is `fizz` defined like this? (We'll write a cleaner version later)

Example Koan - fizzbuzz 2

As an aside, would this work (i.e. without using `funcall`)?

```
...
((if (zerop ...) (lambda (x) ...) #'identity) (buzz num))
```

Example Koan - fizzbuzz 3

Are these better – and which of these is the best?

```
(defun fizz (num)
  (let ((b (buzz num)))
    (if (zerop (mod num 3)) (cons 'fizz b) b)))
```

or

```
(defun fizz (num)
  (if (zerop (mod num 3)) (cons 'fizz (buzz num))
      (buzz num)))
(fizz 7)
nil
(fizz 3)
(fizz)
(fizz 5)
(buzz)
(fizz 15)
(fizz buzz)
```

Example Koan - fizzbuzz 4

Digression: We need to generate lists of integers (called iota from APL, A+ et al):

```
(defun iota (k)
  "Generate a list of integers from 1 to k"
  (labels ((iota1 (k1)
                  (if (< k1 1) nil
                    (cons k1 (iota1 (1- k1))))))
    (nreverse (iota1 k))))
```

Meditations:

- Ponder the use of `nreverse` – why it is needed, why it is safe
  - We shall see later how to build better `iotas`
- Why might the variable/function naming not be ideal? Could we have used `k` in `iota1`? (We will return to naming shortly)
- `(defun fact (k) (reduce #'* (iota k)))`

Example Koan - fizzbuzz 5

This is not true fizzbuzz but we want to show `mapcan` in EL:

```
(defun fizzbuzz (num)
  (mapcan #'fizz (iota num)))
(fizzbuzz 15)
(fizz buzz fizz fizz buzz fizz fizz buzz)
```

The point is that `fizz` can return 0, 1 or 2 results and they are merged into the result list (destructively!)

- `mapcan` merges lists together (destructively), so the map function can return multiple results
- For 0 or 1 values, it may be cleaner to use `delete`, `delete-if` or `delete-if-not` (as appropriate)
  - Note that logically `delete-if-not` does "select-if"
  - `remove`, `remove-if` and `remove-if-not` are the side-effect-free versions

Example Koan – FizzBuzz $-1

Since we've started, we might as well solve FizzBuzz (with the definitions from previous slides):

```
(defun fizzbuzz-number (k)
  "Return symbol from fizzbuzzing a number, or nil"
  (let ((fb (fizz k)))
    (if (cdr fb) 'fizzbuzz  ; >1 elt
      (car fb))))

(defun fizzbuzz (k)
  "FizzBuzz from 1 to k"
  (mapcar (lambda (x) (or (fizzbuzz-number x) x))
    (iota k)))

(fizzbuzz 21)
(1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz ...)
```

Example Koan – FizzBuzz $

Incidentally, in CL if we are worried about the extra list being generated (and eventually gc'ed) in

```
(mapcar (lambda (x) ...) (iota k))
```

we can reuse the list with (permitted) side effect (as the value produced by `iota` is an r-value):

```
(let ((m (iota k)))
  (map-into m (lambda (x) ...) m))
```

This tells Lisp that we want to reuse `m` (on the LHS of the lambda) while mapping the values of `m` (on the RHS), but it obviously less readable than a `mapcar` and requires the `let` binding.

Unfortunately, while EL has a function called `map-into`, it does something quite different.

We will return to fizzbuzz in the Advanced Topics sections 10 and 13.

Advanced maps: Advanced List Iterations

`dolist` iterates over a list. Functionally we have done the same either with `first` and `rest` and *recursion*, or with `mapcar`. The latter is like `dolist` except it produces an output:

```
(mapcar
  (lambda (state) (when (goalp state) (throw 'found state)))
  data)
(nil nil nil nil nil ...)
```

An output list is needlessly produced, as we here are calling only for side effect (more on side effects later).

Lisp has functions which will happily do the same and discard the result of λ:

```
(map nil (lambda (state)
           (when (goalp state)
             (throw 'found state)))
  data)
```

Advanced maps

How does `map` differ from `mapcar`? It works on *sequences*:

```
(map 'list (lambda (x) (+ x 2)) [1 2 3 4])
(3 4 5 6)
(map 'vector (lambda (x) (- x 2)) '(3 4 5 6))
[1 2 3 4]
(map 'string (lambda (x) (+ x 32)) "ABCD")
"abcd"
```

(the latter being EL only) but accepts `nil` as its type to discard the output.

Of course this is cleaner if only the type change is desired:

```
(coerce (list 1 2 3 4) 'vector)
[1 2 3 4]
```

Two patterns implementing mapcar

We could have written EL's (simpler) mapcar like this:

```
(defun our-mapcar (func lst)
  (if (endp lst) nil
    (cons (funcall func (first lst))
          (our-mapcar func (rest lst)))))
our-mapcar
(our-mapcar #'sqrt '(16 4 36 5))
(4.0 2.0 6.0 2.23606797749979)
```

except that it may run out of stack for a longer list (we return to this problem later, in Advanced Topics section 5)

- Indeed much of Lisp can be (or is) constructed from a smaller set of primitives (McCarthy)
- As we have seen before, the first/rest (or car/cdr) pattern is very common
- As is the cons/nil constructing the result
- Usually this (combined) pattern is implemented with mapcar

Advanced maps – sliding window

But maps can do more advanced stuff. Let us first return to the sliding window exercise from the first talk. We want to sum values

```
(sum-window 3 '(1 5 4 1 3 6 2))
(10 10 8 10 11)
(sum-window 2 '(1 5 4 1 3 6 2))
(6 9 5 4 9 8)
(sum-window 4 '(1 2 3))
nil
```

Sliding Window

Let's start by writing a helper function: it is to sum variables from a list:

```
(sum-helper 4 '(2 5 1 3 6 9 1))
11
(sum-helper 1 '(2 5 1 3 6 9 1))
2
(sum-helper 0 '(2 5 1 3 6 9 1))
0
(sum-helper 5 '(1 2 3 4))
nil
```

This example is Interesting because it has two stop conditions: the counter reaching zero, and the list becoming empty.

Meditation: what is `(sum-helper 0 nil)`? Should it be `0` or `nil`?

Sliding Window

```
(defun sum-helper (n data)
  (cond
   ((zerop n) 0)
   ((endp data) nil)
   (t (let ((result (sum-helper (1- n) (rest data))))
        (and result (+ (first data) result))))))
```

Sliding Window

Now we can write a function to call the helper... we can't just do `mapcar` as the function would see only `first` (as with `dolist`) – so, instead, we could almost do:

```
(defun sum-window (k data)
  (if (endp data) nil
    (cons (sum-helper k data)
          (sum-window k (cdr data)))))
(sum-window 3 '(1 5 4 1 3 6 2))
(10 10 8 10 11 nil nil)
```

This works, but obviously it generates extra `nil`s at the end as it has to map across the whole list - so we could write another function to truncate them away.

Notice the `cons`/`rest` pattern again – *nearly* like the `mapcar` pattern

Sliding Window – Advanced maps

The cons-cdr pattern from the previous slide is a very common functional pattern. Lisp has a mapping function that can do the same in a single line:

```
(defun sum-window (k data)
  (maplist (lambda (d) (sum-helper k d)) data))
(10 10 8 10 11 nil nil)
```

maplist cdrs across the list, consing the results:

```
(maplist #'identity '(1 2 3 4 5))
((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))
```

These are successive cdrs of the *same* list:

```
(let ((m (maplist #'identity '(1 2 3 4 5))))
  (eq (cdar m) (cadr m)))
t
```

Sliding Window – Advanced maps

So we either do

```
(defun sum-window (k data)
  (delete nil (maplist (lambda (d) (sum-helper k d)) data)))
```

and we're done; or we do

```
(defun sum-window (k data)
  (mapcon (lambda (d)
            (let ((v (sum-helper k d)))
              (and v (list v))))
          data))
(sum-window 3 '(1 5 4 1 3 6 2))
(10 10 8 10 11)
```

`mapcon` is to `maplist` what `mapcan` is to `mapcar` – it `nconc`s the results; here we use it to delete the `nil` entries.

Mapping for side effect

This call appears to have the side effect of the lambda, but it doesn't – why?

```
(mapcar (lambda (x) (setq x 3)) (list 1 2 3 4))
(3 3 3 3)
```

but in fact it has no effect at all on the original list:

```
(let ((y (list 1 2 3 4)))
  (mapcar (lambda (x) (setq x 3)) y)
  y)
```

Meditation: why are we using `list` to create the list instead of the macro '?

Mapping for side effect

This works, though – why?

```
(let ((y (list 1 2 3 4)))
  (maplist (lambda (x) (rplaca x (1+ (car x)))) y)
  y)
(2 3 4 5)
```

but of course `maplist` still creates a temporary list which is also updated:

```
(let ((y (list 1 2 3 4)))
  (maplist (lambda (x) (rplaca x (1+ (car x)))) y))
(2 3 4 5)
```

The lists are not eq (though their elements are):

```
(let ((y (list 1 2 3 4)))
  (eq y (maplist (lambda (x) (rplaca x 3)) y)))
nil
```

This ends the review of the mapping functions!