Functional Programming 3

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

March 27, 2022

Outline of Talks

- Previous talks (talks 1 and 2):
  - Introibo
  - Pure Functional Programming Principles
  - Mapping
- This talk (talk 3):
  - Labels and naming
  - Lists (or rather cons cells)
- 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 2 – Mapping

One of the core functional programming patterns is to recurse through a list with `cdr` (or `rest`), gathering results of a function being called on the elements.

Lisp's powerful mapping functions implement this fundamental pattern very concisely:

function sees:	list of single value results	nconc'ed list of results	nothing returned
first elt	`mapcar`	`mapcan`	`mapc`
whole sublist	`maplist`	`mapcon`	`mapl`

- Sequence functions (lists, vectors, strings)
  - `map` acts on *sequences*, returning a sequence (of possibly different type) or `nil` (side effect call)
  - `reduce` implements the pattern we used in `fact`

Summary of Talk 2 – Functional Lisp

We saw how core features of Lisp support functional programming:

- `nil` is very powerful:
  - It evaluates to false
    - In python and Perl, the empty list evaluates to false
    - But in Lisp, false is an empty list (see `iota1` below)
  - `(car nil)` $\Rightarrow$ `nil` (and the same for `first`) makes code cleaner, shorter
    - E.g. `(endp (cddr w))` says "`w` has 0-2 elements"
- Using `and` and `or` as control flow
  - `and` returns the first `nil` it finds, or last expr
  - `or` returns its first non-`nil`
  - E.g. `iota1` (helper for `iota`):
    - `(defun iota1 (k1) (and (>= k1 1) (cons k1 (iota1 (1- k1)))))`
    - `(when (>= k1 1) ...)` will also return `nil` for `k1`<1 but the `if` version is better style
- Using convenient *symbols* where Inferior Languages™ use strings, enums, etc (more on symbols in this talk and later talks)

Summary of Talk 2 – Answers

Answers to meditations and exercises:

- `and`, `or`, `if` etc *are not functions* so cannot be used with `funcall` and mapping functions
- How to do `(search m "JanFebMar...")` better
  - Answer later in this talk
- Why do we need to copy a list before sorting it?
  - Because `sort` destroys its argument
  - It is an "`n`" function though not named so
  - The `mapcars` in the Schwarzian transform created temporary lists
- Similarly if we know the list will be altered, we may have to create it as `(list 1 2 3 4)` rather than `'(1 2 3 4)`
  - Lisp may be justified in assuming `'(1 2 3 4)` constant
  - Though ELisp doesn't!
- `labels` and naming
  - Answer on next slide

To labels or not to labels?

```
(defun iota (k)
  "Generate a list of integers from 1 to k"
  (labels ((iota1 (k1)
                  (and (>= k1 1)
                    (cons k1 (iota1 (1- k1))))))
    (nreverse (iota1 k))))
```

or

```
(defun iota-1 (k1)
  "Helper function for iota - generate integers from k1 to
  (and (>= k1 1)
      (cons k1 (iota1 (1- k1)))))
(defun iota (k)
  "Generate a list of integers from 1 to k"
  (nreverse (iota-1 k)))
```

To `labels` or not to `labels`?

- Standalone function
  - Easier to write/debug – and read
  - Can be unit tested (e.g. with `rt` in CL)
  - Can be traced (in CL) or inspected, as it has a persistent name
  - Sometimes the helper needs to be defined before its parent (next slide)
  - Safety: the helper cannot access its parent's lexical scope
- `labels`
  - Safety: Can never be called from `labels`' lexical scope
  - Convenience: Helper is in parent's lexical scope
  - Fewer standalone functions cluttering our code
  - Mutual recursion (adv. top. section 3)

The point is that helper functions, if their parameter types are guaranteed, can optimise those checks away (in principle), regardless of whether they are standalone functions or declared with `labels` (or `flet`).

One approach is to develop as a standalone function, but for smaller functions later embed as `labels`.

Interdependency

Lisp is traditionally generous with letting us define functions as we need them:

```
(defun foo (k)
  (if (oddp k) (bar k) (+ k 5)))
```

This function will cheerfully run as long as we keep passing it even numbers, even though `bar` is not defined. However, some CLs such as SBCL will warn if a function is not defined at read/compile time (SBCL compiles on read).

In turn, we can *stub* `bar` for now, if it is called rarely (EL version):

```
(defun bar (n)
  "FIXME: bar stub"
  (string-to-number
    (read-string (format "bar(%d) = ?" n))))
```

Once we replace `bar` with its production version, `foo` automagically calls that instead.

Lists – Assumed Background

We assume you know your cons cells:

- The cons cell has a `car` and `cdr`
  - `(cons 'x 'y)` ⇒ `(x . y)`
  - `(car '(x . y))` ⇒ `x`
  - `(cdr '(x . y))` ⇒ `y`
- A normal list has the `car`s pointing to the elements
  - The `cdr`s point to the next cons cell
  - In particular, the last cell's `cdr` is `nil`
- The *dot* tells the *Reader* the `cdr` follows:
  - `'(2 . nil)` ⇒ `(2)`
  - `'(2 . (a b))` ⇒ `(2 a b)`
  - The *Printer* will always choose the most concise (and re-readable) representation
- A list whose final `cdr` is not `nil` is an *improper* list
  - These are not allowed in iterations (mapping, recursion) as we almost always test whether the `cdr` is `nil` to stop

More Standard Types with Lists: lookup

Lisp has powerful types built in, that Inferior Languages™ need to pull in from (usually standard) libraries.

While Lisp has hash tables for key/value lookup, in simple cases lists will do perfectly well and have a long history. The *alist* (short for association list) is a standard type in Lisp, yet constructed from standard conses:

```
(let ((dict '((a . 1) (b . 2) (c . 3))))
  (list (assoc 'b dict) (rassoc 3 dict) (assoc 'f dict)))
((b . 2) (c . 3) nil)
```

You have to do `(cdr (assoc ...))` to get the value, or a `(car (rassoc ...))` for the reverse lookup; both return the full cons cell. Meditation: why is this the better choice (next slide)?
If you are not using `rassoc`, the key/value does not have to be a dotted pair.

```
(let ((dict '((a 1 2) (b 2 y) (c . 3))))
  (assoc 'b dict))
(b 2 y)
```

More Standard Types with Lists: lookup

... answering the meditations from the previous slide:

1. We may want to know what the key was:

```
(let ((dict '((1 . a) (2 . b) (3 . c))))
  (assoc-if #'evenp dict))
(2 . b)
```

2. or we may wish to modify the value after the lookup:

```
(let* ((dict '((1 . a) (2 . b) (3 . c)))
       (val (assoc-if #'evenp dict)))
  (when val (rplacd val 'z))
  dict)
((1 . a) (2 . z) (3 . c))
```

3. or distinguish looking up value `nil` vs entry doesn't exist:

```
(let ((dict '((1 . a) (2 . nil) (3 . fred))))
  (list (assoc 2 dict) (assoc 'biff dict)))
((2) nil)              ; (2) is (2 . nil)
```

More Standard Types with Lists: lookup

For modifying the alist, this would also have worked and is a more "modern" version of achieving the same goal:

```
(let* ((dict '((1 . a) (2 . b) (3 . c))))
  (setf (cdr (assoc-if #'evenp dict)) 'z)
  dict)
((1 . a) (2 . z) (3 . c))
```

(note it will change only the *first* entry with an even key) but has the obvious disadvantage that it will explode if the lookup fails (ie. it doesn't "autovivify" in Perl-speak).

In contrast, hashes *do* autovivify:

```
(let ((y (make-hash-table :test #'equal))) ; empty
  (setf (gethash "mimsy" y) 'blop)
  (gethash "mimsy" y))
blop
```

More Standard Types with Lists: lookup

`(acons key val dict)` is short for `(cons (cons key val) dict)`.

To add-or-update an entry, we can simply `acons` the new entry in front (it will always match before the entry it supersedes).

This add-to-front is a useful feature of alists:

```
(flet ((lookup-a (g)
                 "Find entry associated with 'a"
                 (cdr (assoc 'a g))))
  (let ((dict '((a . 1) (b . 2) (c . 3))))
    (list (lookup-a dict)
          (lookup-a (acons 'a 7 dict))
          (lookup-a dict))))
(1 7 1)
```

More Standard Types with Lists: lookup

There is even a multiplayer version of acons called pairlis:

```
(flet ((lookup-a (g)
                 "Find entry associated with 'a"
                 (cdr (assoc 'a g))))
  (let ((dict '((a . 1) (b . 2) (c . 3))))
    (list (lookup-a dict)
          (lookup-a (pairlis '(c a) '(12 13) dict))
          (lookup-a dict))))
(1 13 1)
```

pairlis takes a list of (distinct) keys (which may shadow those in the alist as above) and a list of the corresponding values and returns the alist with those added to the front.

More Standard Types with Lists: lookup

This is potentially useful/interesting because Emacs uses alists – for example, several alists help select the major mode:

```
(length auto-mode-alist)
266
(push (cons "\\.R$" 'text-mode) auto-mode-alist)
(find-file "foo.R") ; opens in text mode
(pop auto-mode-alist)
(find-file "bar.R") ; back to R mode
```

This makes it possible to temporarily override a lookup, although this code is not functional like on the previous slide, as we need to override a system variable (in Advanced Topics section 6, we shall look at a more functional approach)

Exercise: how would we "pop" if the cell is not the first? If the key is b and the alist is `((a . 1) (b . 2) (b . 3))`, we want to remove the b $\Rightarrow$ 2 lookup so we get b $\Rightarrow$ 3. You may (must) modify the alist. There is more than one solution.

More Standard Types with Lists: sets

Lists can also be used as *trees* and *sets*. We show the latter (though these slides contain lots of examples of trees):

```
(let ((y (list 1 2 3 4 3 1 5 2)))
  (delete-duplicates y))
(4 3 1 5 2)
```

(as ever, delete- is the destructive version, remove- creates a fresh list with the result)

```
(let ((x '(1 2 3 4)) (y '(2 4 6 8)))
  (intersection x y))
(4 2)
(let ((x '(1 2 3 4)) (y '(2 4 6 8)))
  (union x y))
(8 6 1 2 3 4)
(let ((x '(1 2 3 4)) (y '(2 4 6 8)))
  (set-difference x y))
(1 3)
```

These have *n* versions if altering their arguments is allowed.

More Standard Types with Lists: sets and lookup

Depending on what your set/dict elements are, you may need to specify the test function:

```
(let ((dict '(("abc" . 1) ("def" . 2))))
  (assoc "abc" dict))
("abc" . 1)
```

In EL, the default test function for `assoc` is `equal` but a test function can be passed in as a third argument. In CL, the situation is different as it uses `eql` by default:

```
(let ((dict '(("abc" . 1) ("def" . 2))))
  (assoc "abc" dict))
NIL
(eql "abc" "abc")
NIL
(let ((dict '(("abc" . 1) ("def" . 2))))
  (assoc "abc" dict :test #'equal))
("abc" . 1)
```

More Standard Types with Lists: sets and lookup

Caution: because the default test function for the set functions is not `equal` even in EL:

```
(remove-duplicates '("abc" "def" "abc" "def" "ghi"))
("abc" "def" "abc" "def" "ghi")
(remove-duplicates '("abc" "def" "abc" "def" "ghi")
  :test #'equal)
("abc" "def" "ghi")
```

`remove-duplicates` was defined in the `cl` package, so it implements the interface of its CL equivalent (including the `:test` keyword). This is unlike `assoc` which is from the core EL, so is different from the CL version.

Exercise: write `remove-duplicates` using (1) recursion and (2) mapping. It should take an equality test function as an optional parameter (extra credit: use the `:test` keyword)

More Standard Types with Lists: sets and lookup

Using alists, we can revisit last talk's exercise that used

```
(search m "JanFebMarAprMayJunJulAugSepOctNovDec")
```

and replace the lookup with an alist:

```
(defun month-number (m)
  (cdr (assoc m '(("Jan" . 1) ("Feb" . 2) ...))))
```

but in CL, the `assoc` needs a `:test #'equal`.

This implementation has additional advantages:

- It does not return valid results for nonsense strings like `"anF"`
- It needs no extra calculations (`(1+ (floor ... 3))`)

Digression: which is faster?

Quick test macro (meditation: why does it have to be a macro?) (code in the reference uses gensyms) (and it's EL only)

```
(defmacro exec-time (expr)
  `(let ((t1 (current-time)))
     (cons ,expr
           (let ((t2 (current-time)))
             (time-subtract t2 t1)))))
exec-time
(exec-time (+ 2 3))
(5 0 0 1 272000)
```

It returns its result followed by timing data (which we could tidy up)

Full code available at `https://github.com/jjensenral/functional/blob/main/code/month.el`

Digression: which is faster?

Results of 10,000 tests (run on the author's old slow laptop):

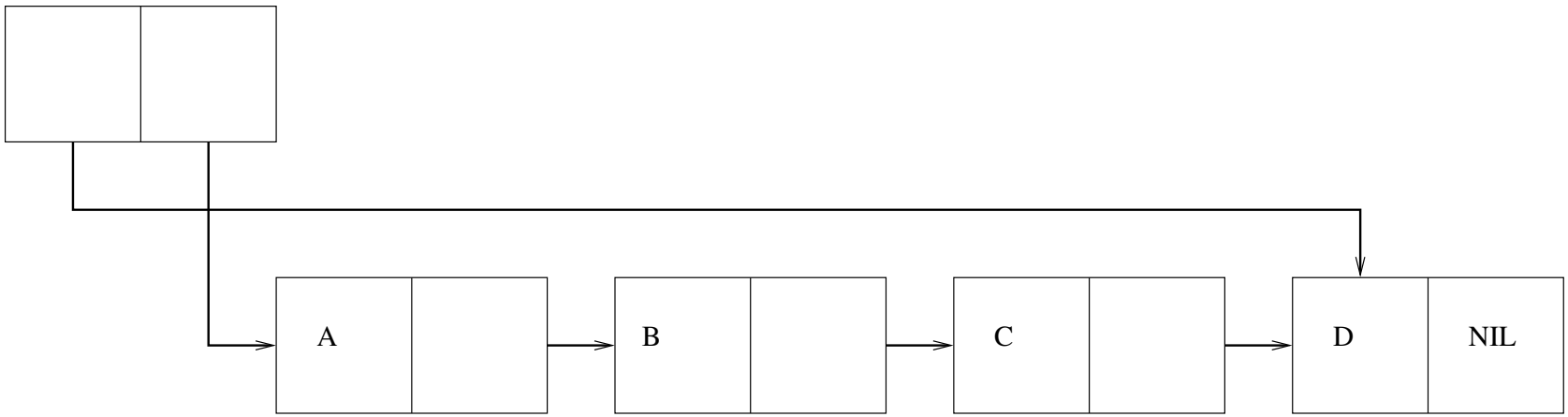
Months are	Lookup	Time (ms)	Rank
Strings	`search`	966	6
Strings	`assoc`	68	5
Strings	`gethash`	66	4
Strings	`position`	28	2
Symbols	`assoc`	65	3
Symbols	`position`	18	1

We can see the search method (from Talk 2) has the worst performance, and symbols generally do better than strings, as we would expect – symbols are atoms, strings are sequences.

More Standard Types with Lists: queues

Lists can `push` and `pop` at the front but not naturally at the back. How would we implement a queue (FIFO)? Could we implement a dequeue?

Lists can be used to implement an efficient FIFO:



- Meditation: why is the `car` of the top cons pointing to the tail and not the `cdr`?
- Exercise: construct such a FIFO from any list
- Exercise: How do we implement enqueue/dequeue?
- Exercise: Can we implement a dequeue (push/pop at both ends)?

Non-standard Types with Lists

1. Improper lists: `(a b . c)`
2. Shared list structure:

```
(let* ((base '((a . 2) (b . 3)))
       (l1 (acons 'c 3 base))
       (l2 (acons 'd 4 base)))
  (list l1 l2))
(((c . 3) (a . 2) (b . 3)) ((d . 4) (a . 2) (b . 3)))
```

3. Infinite lists:

```
(let ((w (list 1 2 3 4)))
  (rplacd (last w) w)
  w)
(1 2 3 4 1 2 3 4 1 2 . #5)
```

Emacs saves the day by abbreviating the output (cf. Adv Top. 6, 10). You can even ask the *Reader* to create the list:

```
'#1=(a b c d . #1#)
(a b c d a b c d a b . #5)
```

Naming and Abstractions

To write clean production code, it may be useful to introduce abstractions. As a simple example, let us return to the *pair* using a `cons`, like we had in our *alists* earlier:

```
(defun make-pair (x y) (cons x y))
(defun pair-first (p) (car x))
(defun pair-second (p) (cdr x))
```

The naming used here is consistent with Lisp *structures* except for the absence of keywords in the lambda list (see next slide).

Naming and Abstractions

Using structures instead, we get more bang for the buck:

```
(defstruct pair first second)
pair
(make-pair :first 'a :second 'b)
#s(pair a b)
(pair-first #s(pair a b))
a
(copy-pair #s(pair a b))
#s(pair a b)
(make-pair :second 'bar :first 'foo)
#s(pair foo bar)
```

Structures automatically define a *constructor* and the `#s` *reader macro*, a *copier*, and *accessors* for every slot.
Notice the use of *keywords* in the constructor.

Digression: Compatibility notes: Lambda Lists

EL does not support full CL lambda lists, so this is CL:

```
(defun make-pair (&key first second) (cons first second))
(make-pair :second 2 :first 1)
(1 . 2)
```

In EL we could fudge this with

```
(defun make-pair (key1 x key2 y)
  (unless (and (eq key1 :first) (eq key2 :second))
    (error "Invalid keywords"))
  (cons x y))
(make-pair :first 1 :second 2)
(1 . 2)
```

though it would still rely on `:first` being first; fixing this is left as an exercise to the reader/listener (what if there were 3 keywords?).

EL supports `&optional` (but default is always nil, and there is no supplied-p) and `&rest`.

Naming and Abstractions

We can then write auxiliary functions which do not need to know how the pair was implemented (this one is using the struct but it could have worked with the cons (keywords version)):

```
(defun zip-pairs (list1 list2)
  (if (and list1 list2)
      (cons (make-pair :first (first list1)
                       :second (first list2))
            (zip-pairs (rest list1) (rest list2)))
    nil))
zip-pairs
(zip-pairs '(a b c) '(1 2 3))
(#s(pair a 1) #s(pair b 2) #s(pair c 3))
```

This is like python's "zip" (not to be confused with CMUCL's python which predates python). Of course in CL, one would not bother with such trivialities, but merely write `(mapcar #'cons list1 list2)` or `(pairlis list1 list2 nil)`.

Naming and Abstractions – Summary

Arguably the pair is a quite natural fit to `cons`; but when implementing *trees* using native Lisp types, it may help to have an abstraction, because there is more than one way to do it and the choice may depend on how the tree is used.

- The *disadvantage* of the abstraction is the extra function call
  - Although CL at least can *inline* functions, or optimise the extra call away.
  - And also writing more code; inlining doesn't help with that...
  - Though CL's built in generator might, theoretically (`#.` reader macro)
  - Possibly less optimisation across functions than inside one?
- Norvig's PAIP is a good place to look for Useful Code using clean abstractions (in CL, not EL);
- Abstractions can also introduce extra *type checks* during development, or *stubs* for complicated stuff;
- While EL is more limited, CL has a very powerful type system which can implement checks (adv. top. sec. 7)

Summary - the Lisp superpowertools

Summarising the Lisp functional programming Super Power Power Tools we have met (plus one from an earlier talk):

- *Recursion*, the elegant engine of functional programming;
- `list` and list tools, designed for the first/rest paradigm;
- `cond` and friends for dividing and conquering;
- Mapping functions (and `reduce`) implement functional patterns concisely (including recursion, without recursing)
- Sequence functions including `reduce`
- Functions: `funcall`, `lambda`, `apply`; multivalued in CL
- `lambda` lists in Lisp are much more powerful than other functional languages – more powerful than other languages;
- `flet` and `labels` for defining local functions;
- *macros*, because Lisp macros are superpowertools for everything
- *stubs*, and `trace` (the latter CL only)

Topics for Future Talks

1. Lambda (anonymous (unnamed) functions) and *currying*
2. List comprehension
3. Functions – mutually recursive, higher order
4. Symbols
5. Tail recursion
6. Scope and extent (Lisp)
7. Types and type inference
8. Branch-on-pattern-matching and guards
9. Memoisation
10. Lazy evaluation types
11. Pipes style composition
    - $h(g(f(x))) \equiv$ `(h (g (f x)))` $\equiv x|f|g|h$
12. Monads: theoretical framework for types and computation
13. Applied monads: Maybe, Arrays
14. Bonus section for survivors of MonadLand: Lisp Hacking