Functional Programming 4

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

April 24, 2022

Outline of Talks

- Previous talks (talks 1-3):
  - Introibo
  - Pure Functional Programming Principles
  - Mapping
  - Labels and naming
  - Lists
- This talk (Talk 4):
  - 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.

Superpowertools – Lemonodor fame awaits!

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 such as `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, but see later)

Common/Advanced(ish) Features of Functional Languages

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 (not the lazy kind) 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

Common Features of Functional Languages - Currying

- Currying (binding some but not all function arguments)
  - In this example + is curryed, binding its second argument to 2
    - (Assuming + is considered as a function of *two* args (which it isn't))
  - "Proper" functional languages like F# do this more elegantly
    - As long as we bind the *first argument*
    - E.g. + 2 would be the function that adds two to something

```
(mapcar (lambda (x) (+ x 2)) '(1 2 3 4))
(3 4 5 6)
```

- Hence the *type* notation of a function $+ :: \text{int} \rightarrow \text{int} \rightarrow \text{int}$
- After currying the (first) argument, the expr has type $:: \text{int} \rightarrow \text{int}$
- Of course in Lisp, `#'+` takes any number of arguments (incl 0)
- In talk 2 we had `(apply #'mapcar #'list rows)` where `apply` effectively curryes `mapcar` by binding `#'list` as its first argument

Lambdas

Almost but not quite a translation of λ calculus. Taking the Quine

$$((\lambda x.(xx))(\lambda x.(xx)))$$

becomes in EL

```
((lambda (x) (list x x)) (lambda (x) (list x x)))
((closure (t) (x) (list x x)) (closure (t) (x) (list x x)))
```

which is a Quine if you appreciate that evaluating `lambda` coerces the `lambda` expression into a function

```
(lambda (v) (+ v 2))
(closure (t) (v) (+ v 2))
(coerce (lambda (v) (+ v 2)) 'function)
(closure (t) (v) (+ v 2))
```

Lambdas

In CL, functions can return multiple values (including none) but special tools are needed to capture the not-first values:

```
(floor 12 7)
1
5
(multiple-value-bind (q r) (floor 12 7)
  (+ (* q 7) r))
12
```

When not used in a multiple value context, only `floor`'s first value is used (as in EL where it returns only one value):

```
(+ (floor 12 7) 3)
4
```

Lambdas

Both `lambda` and `let` create variable bindings:

```
((lambda (v)
   (+ v (let ((v 2)) (* 3 v))))
   6)
12
```

- The inner `v` is bound by `let` to the value 2
- The outer `v` is bound by `lambda` to the value 6
- It's the same symbol `v` but the inner binding shadows the outer during execution of the `let`
- The outermost parenthesis

List comprehension

List comprehension is the idea of constructing elements

$$(f(x, y, \ldots)|x \in A, y \in B(x), \ldots)$$

analogous to how sets are constructed (this is obviously not a precise definition but a it-will-do-for-now expression of the idea).

Arguably, list comprehension for Lisp is `loop`, even though it looks imperative sometimes:

```
(loop for i from 1 to 10 collecting (* i i))
(1 4 9 16 25 36 49 64 81 100)
```

but this could be done with `iota` functionally. It can express actions on lists (and vectors) equally naturally:

```
(loop for a in '(1 2) nconcing
  (loop for b in '(3 4) collecting (cons a b)))
((1 . 3) (1 . 4) (2 . 3) (2 . 4))
```

List comprehension

Another common case is calling a function (with side effects!) *n* times, collecting the results (which means we can't just use `dotimes`):

```
(loop repeat 10 collect (random 10))
(4 9 1 2 9 3 8 0 0 8)
```

The alternative could be

```
(map 'vector (lambda (x) (random 10)) (make-vector 10 0))
[8 3 3 9 8 6 5 0 5 3]
```

which would have made more sense with CL's `map-into`

```
(let ((v (make-array 10)))
  (map-into v
    (lambda (x) (declare (ignore x)) (random 10))
    v))
```

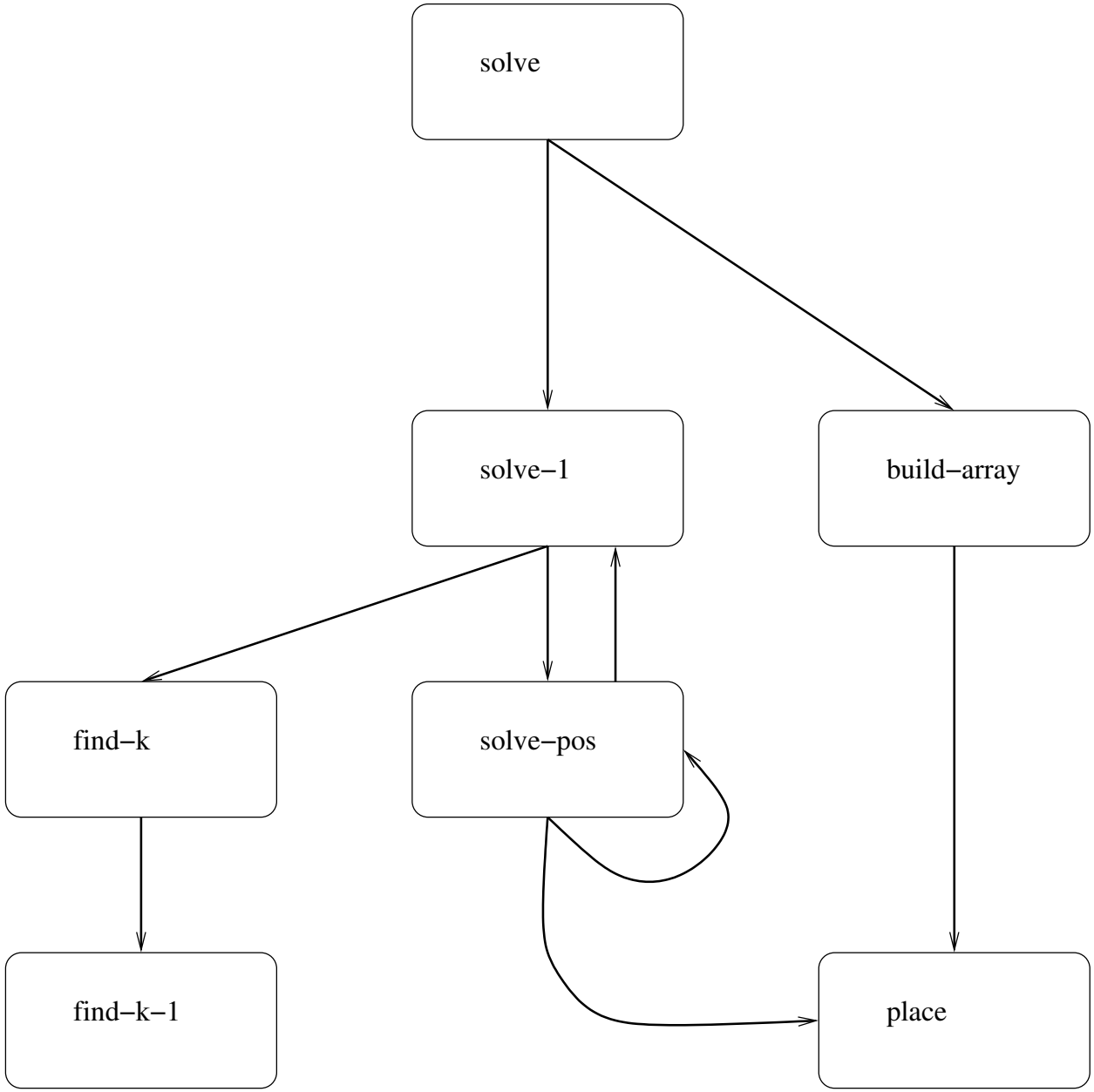
Mutually recursive functions

A puzzle: given a number N, create an array of length $2N$ with each of the digits $1, \ldots, N$ occurring precisely twice, such that for every digit $k \in \{1, \ldots, N\}$ there is k digits between the two occurrences of the digit k.

Example for $N = 3$: `312132`. Starting digits may be given, e.g. starting with `*1*1**`

Solving it functionally, the original design had three functions calling each other – now we're down to two...

Functions – Mutually recursive functions



Functions – Hooks and callbacks

Getting one value out of a deeply nested computation: throw it.

```
(... (when (goalp state) (throw 'found state)) ...)
```

(for some hypothetical function `goalp`.) When we need to collect multiple values, a callback makes sense:

```
(... (when (goalp state) (funcall cb state)) ...)
```

Later we shall look at other ways of solving the same problem.

Functions – Hooks and callbacks

Suppose for a moment we want Fibonacci numbers (to which we will also return later), starting with the CL version:

```
(defun fib (k &optional (a 0) (b 1))
  (if (zerop k) b
    (fib (1- k) b (+ a b))))
FIB
(fib 10)
89
```

Functions – Hooks and callbacks

In CL we can now do

```
(trace fib)
(FIB)
(fib 5)
  0: (FIB 5)
    1: (FIB 4 1 1)
      2: (FIB 3 1 2)
        3: (FIB 2 2 3)
          4: (FIB 1 3 5)
            5: (FIB 0 5 8)
            5: FIB returned 8
          4: FIB returned 8
        3: FIB returned 8
      2: FIB returned 8
    1: FIB returned 8
  0: FIB returned 8
8
(untrace fib)
```

Functions – Hooks and callbacks

In EL, the `fib` function could look like this:

```
(defun fib (k &optional a b)
  (if (zerop k) (or b 1)
    (fib (1- k) (or b 1) (+ (or a 0) (or b 1)))))
fib
(fib 10)
89
```

Functions – Hooks and callbacks

EL uniquely allows us to alter functions with *advice*:

```
(defun my-trace (name func args)
  ;; Enter function
  (princ (format "%s: %s\n" name args))
  ;; Call function and remember value
  (let ((val (apply func args)))
    ;; Print result
    (princ (format "%s => %s\n" name val))
    val))
(advice-add 'fib :around
  (lambda (func &rest args) (my-trace 'fib func args))
  '((name . trace)))
nil
```

In this respect, advice work quite a lot like methods (specifically, standard method combination) in CLOS

Functions – Hooks and callbacks

```
(fib 5)
fib: (5)
fib: (4 1 1)
fib: (3 1 2)
fib: (2 2 3)
fib: (1 3 5)
fib: (0 5 8)
fib => 8
fib => 8
fib => 8
fib => 8
fib => 8
fib => 8
8
(advice-remove 'fib 'trace)
nil
```

Functions – Hooks and callbacks

Emacs uses *hooks* a lot, callbacks called at specific times. Normal hooks are single function or (more or less) lists of functions to be called without arguments:

```
text-mode-hook
(text-mode-hook-identify)
(add-hook 'text-mode-hook 'auto-fill-mode)
(auto-fill-mode text-mode-hook-identify)
c++-mode-hook
(irony-mode)
(add-hook 'c++-mode-hook
  (lambda () (message "Hack and be merry!"))
  10)
(irony-mode (closure (t) nil (message "Hack and be merry!")))
```

Functions – Higher order functions

```
(defun compose (f g)
  "Compose two functions calling g first then f"
  (lambda (&rest data) (funcall f (apply g data))))
compose
(funcall (compose #'sqrt #'+) 2 3 4)
3.0
(mapcar (compose #'sqrt #'abs) '(-1 -4 9 -16))
(1.0 2.0 3.0 4.0)
```

Functions – Higher order functions

The CL standard (ANSI X3J13) has tried to deprecate the `-if-not` functions in favour of the `-ifs`:

```
(remove-if-not #'evenp '(1 2 3 4 5 6))
(2 4 6)
(remove-if (complement #'evenp) '(1 2 3 4 5 6))
(2 4 6)
```

The first of these works fine in EL but EL does not have `complement`. However, we can easily write it:

```
(defun complement (func)
  "Logical complement of a function of any number of args"
  (lambda (&rest args)
    (not (apply func args))))
```

Notice that no `funcall` is required when we use it (why not?)

Symbols

Symbols are one of the important tools of Lisp, as

- Names – of variables, functions, catches, types, structures, classes, etc.
- Enums (eg. `(list 'Jan 'Feb 'Mar ...)`)
- Keys for hooks or looking up stuff
- Anywhere else where you need an atom

What is going on here (this is EL but could be CL with one difference):

```
(atom nil)
t
(eq 'fred 'Fred)
nil
(eq 'fred 'fred)
t
(eq (make-symbol "fred") (make-symbol "fred"))
nil
```

Symbols

A symbol has precisely three things:

- A *name* (or more accurately, a *print name*):

```
(symbol-name 'fred)
"fred"
(symbol-name (make-symbol "wilma"))
"wilma"
```

- A *package* (sort of, EL doesn't really do packages)
- A *property list*:

```
(symbol-plist 'fred)
nil
(symbol-plist 'car)
(byte-compile byte-compile-one-arg byte-opcode byte-car ...
... byte-optimizer byte-optimize-predicate
    side-effect-free t)
```

Symbols

If we take the plist first, it provides key/value lookup. Each symbol has its own plist.

```
(get 'fred 'foo)
nil
(setf (get 'fred 'foo) 'blemps)
blemps
(get 'fred 'foo)
blemps
(setf (get 'fred 'bar) 17)
17
(symbol-plist 'fred)
(foo blemps bar 17)
```

It works exactly like an alist except

- for hysterical raisins the structure is different
- it returns the value rather than the key/value
- it does not normally shadow
- it has fewer lookup functions (no equiv of `assoc-if`)

Symbols

Unlike CL, Emacs uses property lists extensively:

```
(get 'sort 'side-effect-free)
nil
(get 'mapcar 'side-effect-free)
nil
(get 'cons 'side-effect-free)
error-free
(get '+ 'side-effect-free)
t
```

Keys should normally be symbols, though values need not be

```
(setf (get 'fred "name") "jones")
"jones"
(get 'fred "name")
nil
```

Symbols

Removing a key ("tag") from a plist is slightly different:

```
(remprop 'fred 'foo)
t
(symbol-plist 'fred)
(bar 17)
```

Altering plists of EL's built in functions is not advisable!

There are functions to work directly on plists:

```
(let ((plist (list 'a 1 'b 2)))
  (setf (getf plist 'c) 3)
  (remf plist 'b)
  plist)
(c 3 a 1)
```

Symbols

Let's return to the packages. In simplified terms, a package is a context where symbols can be looked up; when the Reader first reads a symbol it is created:

```
(eq 'fred 'fred)
t
```

The same symbol is referenced twice, so obviously eq. However, a small number of functions and macros create symbols that are *not in any package*, so cannot be looked up again:

```
(make-symbol "fred")
fred
(eq (make-symbol "fred") 'fred)
nil
```

Here `'fred` is in the package but `make-symbol` creates one that isn't, so they can never be eq: effectively, symbols not in the (same) package cannot be eq even if they have the same name.

Symbols

We use this construction to write macros (this is CL from the author's Advent of Code 14/12/2021):

```
(defmacro incf-list2-entry (list x y delta)
  "Create or add delta to the entry in list for the pair x y
  (let ((bzz (gensym)) (garg (gensym)))
    `(let* ((,bzz (cons ,x ,y))
            (,garg (assoc ,bzz ,list :test #'equal)))
       (if ,garg
           (incf (cdr ,garg) ,delta)
           (setf ,list (acons ,bzz ,delta ,list)))
       ,list)))
```

(For EL code, leave out the `:test #'equal`). Dodgy names apart, the macro is a variation on `incf` for alists of the form

```
(((C . B) . 1) ((N . C) . 1) ((N . N) . 1))
```

– looking up a pair `(x . y)`, it creates or adds to the pair.

Symbols

The Reader can also create symbols not-in-any-package using Reader macros:

```
(eq '#:fred 'fred)
nil
(eq '#:fred '#:fred)
nil
(symbol-name '#:fred)
"fred"
```

The Reader macro `#:` creates a symbol with the same print name but it is in no package; the next time the Reader reads the same characters, a different symbol is created.

Symbols

Since it is created by the Reader (before Eval), we need to use other Reader constructions to reference the symbol (EL code):

```
(defmacro incf-list2-entry (list x y delta)
  `(let* ((#1=#:bzz (cons ,x ,y))
          (#2=#:garg (assoc #1# ,list)))
       (if #2#
           (incf (cdr #2#) ,delta)
           (setf ,list (acons #1# ,delta ,list)))
       ,list))
incf-list2-entry
(let ((mylist (list '((a . b) . 3))))
  (incf-list2-entry mylist 'a 'b 2))
(((a . b) . 5))
(let ((mylist (list '((a . b) . 3))))
  (incf-list2-entry mylist 'x 'y 17))
(((x . y) . 17) ((a . b) . 3))
```

Symbols

Now let's return to symbols that *are* in "the" package, called *interned* symbols (symbols not in a package are called *uninterned*):

```
(eq 'fred (intern "fred"))
t
(eq (intern "fred") (intern "fred"))
t
```

Symbols are atoms and not sequences; if we really wanted to turn a list of two elements `(fizz buzz)` into a single symbol `fizzbuzz`, we'd have to do (EL code)

```
(defun join-symbols (w)
  (if (endp (cdr w)) ; 0-1 elt
      (first w)
    (intern (concat (symbol-name (first w))
                    (symbol-name (second w))))))
join-symbols
(join-symbols '(fizz buzz))
fizzbuzz
```

Symbols

Symbols – whether interned or not – have few restrictions on their names: this is a single symbol:

```
(symbolp 'b^2-4*a*c/2*a)
t
```

but notice we cannot add parentheses for the denominator as the Reader would start constructing a list. Likewise, `1-` is a symbol, the name of the function

```
(symbol-function #'1-)
#<subr 1->
```

Also note that in CL (but not in EL), the Reader maps all symbols to upper case:

```
* (eq 'fred 'FRED)
T
* 'fred
FRED
```