Functional Programming 6

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

March 5, 2023

Outline of Talks

- Previous talks (talks 1-3):
  - Introibo
  - Pure Functional Programming Principles
  - Mapping
  - Labels and naming
  - Lists
- This talk (Talk 6):
  - Advanced(ish) Topics (continued)
- 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.

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. **Closures: scope and extent**
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

Back to basics: what is a closure?

```
(let ((a 1) (b 0))
  (defun fib ()
    (psetq a b b (+ b a))
    b))
fib
(fib)
1
(fib)
1
(fib)
2
(fib)
3
(fib)
5
(fib)
8
```

Back to basics: what is a closure?

```
(let ((a 1) (b 0))
```

- `a` and `b` are declared locally, initialised to a pre-step of Fibonacci (OEIS A000045)

```
 (defun fib ()
```

- The function `fib` is declared *inside* the `let` (not at the top-level)

```
   (psetq a b b (+ b a))
   b))
```

- A single step is performed in Fibonacci with value in `b`
- The function remembers its place in the sequence
- Though `a` and `b` are no longer accessible (only through `fib`), they persist (as long as `fib` exists)

Fibonacci generator, OOP version

```
; SLIME 2.27
CL-USER> (defclass fib-gen ()
           ((a :initform 1 :type unsigned-byte)
            (b :initform 0 :type unsigned-byte))
           (:documentation "Fibonacci generator class"))
#<STANDARD-CLASS COMMON-LISP-USER::FIB-GEN>
CL-USER> (defgeneric next (obj)
           (:method ((obj fib-gen))
             (with-slots (a b) obj
               (shiftf a b (+ a b))
               b)))
#<STANDARD-GENERIC-FUNCTION COMMON-LISP-USER::NEXT (1)>
CL-USER> (let ((o (make-instance 'fib-gen)))
           (list (next o) (next o) (next o) (next o)))
(1 1 2 3)
CL-USER>
```

Disclaimer

(The rest of) this talk is about scope and extent, and will be quite technical. It will (hopefully) tie up some loose ends from previous talks and lay a foundation for topics in future talks.

Suggested answers to exercises at the end

Scope and Extent

Quick reminder about the differences between *bindings* and *assignment*.

Assignments (modifying a value) are usually made with `setq` or `setf` (and avoided as a matter of principle in Functional Programming); it overwrites the value held by the variable. `makunbound` removes the value, though the *symbol* still exists.

```
(let ((a 4))
  (let ((a 3) (b a) c)
    (makunbound 'a)
    (unless (boundp 'b) (list a b c))))
(3 4 nil)
```

There is something Interesting™ going on: all variables are in fact bound (`c` to `nil`), but only within the `let` construct, not globally (which is where `boundp` looks): we shall investigate in this section.

Scope and Extent

As preparation, let us zoom in on the RE parts of the REPL of a simple expression:

```
(func foo)
```

In fact, we shall zoom in on the variable part (the function part being similar).

- We assume `foo` names a variable
- We assume `func` names a function

Scope and Extent

Step 0: The Reader reads the parentheses and two symbols, `func` and `foo` and constructs a list containing these two symbols.

```
(list (intern "func") (intern "foo"))
```

- The happens at read-time; if running interactively, it will be read after we hit return
- Note that the Reader is not allowed to use `make-symbol` (here), why not?
- The Reader creates the symbols `func` and `foo` (CL: in the current Package) if they do not already exist
  - Initially, the symbols are *unbound*

Scope and Extent

Step 1: (Evaluate) The symbol `foo` is looked up as a *reference* to a variable

```
foo
```

- For Lisp2s, `foo` is looked up in the variable namespace (in the current package)
  - `func` is looked up in the function namespace
  - There may be type restrictions on `foo` (Section 7)
- The name is looked up in the appropriate context:
  - The variable may be unbound
  - The variable may be bound locally (with `let`)
  - There may be a "global" value (to be defined shortly)

Scope and Extent

Step 2: The *bindings* of `foo` are looked up in the appropriate context (local/global), the *innermost* binding is found

```
(let ((foo 7))
  (let ((foo 'a))
    (func foo)))
```

- The variable may be unbound (have no binding) – which would be an error
- The innermost binding is often the most recently established

Scope and Extent

Step 3: If the binding has a *value* – an object – it is returned and passed to `func` as an argument

```
(func 'a)
```

- A (lexical) binding will always have a value:
  - It is not possible to create a (lexical) binding that does not have a value
  - It is not possible to remove the value from a (lexical) binding:
  - `makunbound` has no effect whatsoever
  - `setq` must assign it a value
- However, there are *dynamic* bindings, too (which *can* be unbound): we shall examine these shortly
- And constants, of course, always have values
  - System constants (like `pi` or `nil` or `long-float-negative-epsilon`) may not be changed (assigned to, or bound)
  - User-defined constants may grudgingly be redefined
- *Inside* the function, the value is bound to the parameter in the function's lambda list

Scope and Extent

Step 4: Bind the function's arguments

```
(defun func (x) (list x x))
```

When the Evaluator calls the function, it creates a lexical binding of the function's parameter(s) to its argument(s).
Once the binding(s) are created, the function's body is evaluated as if it were

```
(let ((x 'a))
  (list x x))
```

Obviously, if there is an outer (prior) binding of `x`, it is shadowed.
Notice the same effect is achieved by

```
((lambda (x) (list x x)) 'a)
```

Scope and Extent

Suppose we are writing a program to solve problems in integer arithmetic (we assume we have egcd, given a, b it finds x, y s.t. $ax + by = \mathrm{gcd}(a, b)$):

```
(defun mod-inv (m k)
  "Invert k mod m (if m k coprime)"
  (let ((a (first (egcd x m))))
    (if (minusp a) (+ a m) a)))
(defun mod-* (m a b)
  "Multiply a and b mod m"
  (mod (* a b) m))
(defun mod-/ (m a b)
  "Divide a by b mod m"
  (mod-* m a (mod-inv m b)))
(defun coprimep (x y)
  "Determine whether two integers are co-prime"
  (= 1 (gcd x y)))
```

Scope and Extent

If the modulus is constant (known at compile time), it would make sense to curry all the arithmetic functions:

```
(defconst +mod+ 17 "common modulus")
(defun mod-inv (k)
  "Invert k mod +mod+ (if +mod+ k coprime)"
  (let ((a (first (egcd x +mod+))))
    (if (minusp a) (+ a +mod+) a)))
(defun mod-* (a b)
  "Multiply a and b mod +mod+"
  (mod (* a b) +mod+))
(defun mod-/ (a b)
  "Divide a by b mod +mod+"
  (mod-* +mod+ a (mod-inv b)))
```

In CL, constants would be defined by `defconstant`
Note π is called `pi`, not `+pi+`; similarly for other built-in constants like `most-negative-fixnum`

Scope and Extent

CL distinguishes between

- Constants – should not be changed by user or program
- Parameters given to the program (but unchanged by the program) (`defparameter`)
- "Local" variables (and functions)
- "Global" functions (and variables)

It is considered bad practice to just `setq` a variable into existence (though EL permits it); instead, `defvar` should be used:

```
(defvar *dims* '(2 3 3) "Dimensions of data array")
```

This is analogous to `defun` for creating (global) functions (more or less), where `flet` is the function analogue to `let` for creating local (lexical) functions/variables.

Apart from encouraging a documentation string, variables introduced with `defvar` have special magic...

```
(special-variable-p 'auto-mode-alist)
t
```

Scope and Extent

If the modulus is constant-ish – it can be changed by the user or the program but is the same across calls to all of the modulus functions (it would usually not make sense mathematically if it weren't) – it can be declared globally:

```
(defvar *mod* nil "Modulus for all mod- functions, initially
*mod*
(special-variable-p '*mod*)
t
(defun mod-+ (a b)
  "Add numbers modulo *mod*"
  (mod (+ a b) *mod*))
```

The CL convention is to use asterisks in the name (e.g. `*random-state*`, `*print-circle*`), though Emacs does not follow this convention.

Scope and Extent

Once defined, the variable can be *assigned* to (as indeed it must in our example), prior to the first call:

```
(setq *mod* 17)
(mod-+ 12 14)
9
```

However, it can also be bound (in this example, it is still 17 from above):

```
(let ((*mod* 11))
  (mod-+ 12 14))
4
(mod-+ 12 14)
9
```

Notice that `*mod*` is not used anywhere in the lexical scope of the `let` binding...

Scope and Extent

- *Scope* is about the *region of visibility* of a variable binding
  - In *lexical scope*, binding is visible within the code block
  - In *indefinite scope*, a binding is visible anywhere
- *Extent* is about the *duration* of a variable binding
  - In *indefinite extent*, a binding is held indefinitely
    - Until the compiler (or GC) can prove it is no longer reachable
  - In *dynamic extent*, a binding is valid only while execution is within the range of the binding

Scope and extent affect not just variable bindings, but everything that can be "looked up" with a symbol: functions, non-local exits, blocks, tags, etc.

Scope and Extent

A `let` binding has *lexical scope* and *indefinite extent*:

```
(let ((m 3))
  (lambda (x) (+ x m)))
```

The value of `m` is accessible only inside the `let`. The resulting lambda expression will "remember" the `m`= 3 binding even though `m` is not accessible to anyone once the `let` is exited. In other words, *closures* work with lexical scopes – and indefinite extent.

In contrast, `*mod*` has *indefinite scope* (can be accessed by any of the `mod-` functions) but *dynamic extent* (as witnessed by the temporary `*mod*`=11 binding). Such variables are also called (and declared as) *special variables*. In EL, the `let` binding shadows the outer value, but the binding remains special.

The combination of indefinite scope and dynamic extent is sometimes called "dynamic scope" (even in the ELisp documentation).

Scope and Extent

We can now return to our Emacs alist example from Talk 3:

```
(let ((auto-mode-alist
        (acons "\\.R$" 'text-mode auto-mode-alist)))
  (find-file "/home/jens/projects/stats/line.R"))
```

This example temporarily binds the (already) special variable `auto-mode-alist`, to a new value shadowing the existing value of R-mode. Note it does not need `declare special` in the binding – why not?

- The file, if it exists, is loaded in text mode.
- During the load, the mapping to text mode is visible to any part of Emacs.
- The local `auto-mode-alist` *shadows* the global one
- Inside the new alist, the new cons cell *shadows* the existing one
- After the `let` is exited (*even in error*), `auto-mode-alist` retains its normal value (via the original binding)
- Any files whose names end with .R will subsequently load in R mode.

Scope and Extent

- Global (Lisp) functions are (essentially) constant – with indefinite extent
- User-defined functions:

```
(defun foo (x) "add two" (+ x 2))
foo
(defun bar (y) "call foo" (foo y))
bar
(flet ((foo (x) (* x 3)))
  (bar 6))
```

What is happening here – what does the `flet` return?

- `foo` is defined to add two to a number
- `bar` is defined to call `foo`
- The `flet` normally creates a *lexically* bound function
- However, there is already a global binding of `foo` to a function definition
- So the question is: does `bar` call the global `foo` (returns 8) or the one defined by `flet` (returns 18)?

Scope and Extent

In CL we can also have "local" variables with indefinite scope and dynamic (non-indefinite) extent:

```
(let ((a 4))     ; CL code
  (declare (special a))
  (foo))
```

Now, while `foo` executes, a is bound to 4, despite `foo` being defined outside of the lexical scope of the `let`. The binding ceases to exist after the `let` is exited. While `let` would normally declare variables with lexical scope and indefinite extent, the declaration changes both so the scope of a is indefinite and the extent is dynamic.

By convention, special variables – whether declared as above, or globally with `defvar` (or by other means beyond this talk) – have names that begin and end with '*', .e.g. `*a*`

Scope and Extent

What happens here (this is CL code):

```
(let ((*special* "foo"))
  (declare (special *special*))
  (let ((*special* 'bar))
    (makunbound '*special*)
    (ignore-errors (list 1 *special* 2))))
```

Contrast with this:

```
(let ((*special* "foo"))
  (declare (special *special*))
  (let ((*special* 'bar))
    (declare (special *special*))
    (makunbound '*special*)
    (ignore-errors (list 1 *special* 2))))
```

The `declare special` does what `defvar` does (in terms of setting the extent) but there are two important differences...

Scope and Extent

```
; SLIME 2.26.1
CL-USER> (defvar *zut* 'baz)
*ZUT*
CL-USER> (defun foo () (list *zut*))
FOO
CL-USER> (foo)
(BAZ)
CL-USER> (let ((*zut* 'bzzt)) (foo))
(BZZT)
CL-USER>
```

`defvar` is stronger than a local special declaration: `*zut*` is special even in the local binding, despite not being declared special. In EL, `defvar` is currently the only option, and bindings behave similarly (like we saw with `auto-mode-alist`)

Scope and Extent

So far we have met

- Lexical scope and indefinite extent – bindings created with `let`, `flet`
  - Except if they shadow a global/dynamic scope
- Effectively "global" objects defined with `defvar` and `defun` have indefinite scope and indefinite extent
  - Though formally they have dynamic scope – the extent being the entire runtime
  - Constants – and built-in functions – also have indefinite scope and indefinite extent
- Catches have dynamic scope – catches are valid only within the extent of the (implicit) `progn` they enclose
  - And variables declared `special` (in CL only)

```
(let ((a 3))       ; EL code
  (declare (special a))
  (special-variable-p 'a))
nil
```

So the remaining question is: does anything have lexical scope and dynamic extent?

Scope and Extent

Blocks have lexical scope and dynamic extent:

```
(block sknomz (list 1 2 3)
              (return-from sknomz (list 8 9))
              (list 4 5 6))
(8 9)
```

The implication is that this is an error:

```
(defun fact ()
  (lambda (n)
    (labels ((fact-1 (k)
                 (when (zerop k) (return-from fact 1))
                 (* k (fact-1 (1- k)))))
      (fact-1 n))))
(funcall (fact) 12)
```

If we write `(return-from fact-1 1)` then it works – in CL. In EL, it still doesn't work though.

Scope and Extent–Closures

Let's look a bit more closely at bindings. What does this construction return?

```
(eq (lambda (foo) (+ foo 2))
    (lambda (foo) (+ foo 2)))
```

In fact it returns `nil` when evaluated in Emacs, but a compiler would be free to optimise and make the lambdas eq.

Would this be optimisable?

```
(let ((foo (list 1 2)))
  (eq (lambda (x) (cons x foo))
      (lambda (x) (cons x foo))))
```

(answer on the next slide)

Scope and Extent–Closures

The compiler would be allowed to optimise the two lambdas and make them the same object (e.g. if the lambdas were returned in a list, they could be eq).

In contrast, these lambdas must be different objects:

```
(list (lambda (x)
        (let ((foo (list 1 2))) (cons x foo)))
      (lambda (x)
        (let ((foo (list 1 2))) (cons x foo))))
```

Meditation: why?

Scope and Extent

This is really an advanced-squared topic so don't worry too much about it: but it is possible to have variables which, like blocks, have both lexical scope and dynamic extent?

To do that, we declare a lexical variable but tell the compiler to give it dynamic extent (CL only):

```
(let ((a (list 1 2 3)))
  (declare (dynamic-extent a))
  (length a))
```

What this tells the compiler is that the binding will not be referenced beyond the duration of the `let`: which is true, here the binding is used only as long as the list is created.

This allows the compiler to (optionally, possibly) optimise the code and allocate a on the stack instead of the heap. Thus, this would be particularly useful in a function definition when performance is important and the compiler cannot deduce that the binding does not need indefinite extent.

Scope and Extent – A Simplified Summary

- *Scope* is about the *region* where a binding is accessible;
- *Extent* is about the time during execution where a binding is accessible;
- `defvar` creates "global" variables
  - These are *special*: indefinite scope and dynamic extent (aka "dynamic scope")
  - Though the "dynamic" is (usually) the duration of the entire program (when created with `defvar`)
  - (Unless the binding is removed with `makunbound`)
- `defun` does the same with functions
  - `fmakunbound` removes the function definition
- `let`/`let*`/`flet`/`labels` create lexical bindings
  - Unless the variable is already special or (CL) is declared special
  - These have lexical scope and indefinite extent ("lexical binding")
  - Closures rely on these: the indefinite extent is needed to continue to access the binding
  - A lexical binding can *never* be unbound

Answers to exercises

A collection of hacks showing possible answers to the exercises...
and another summary

Answers – Variables

This is CL code (also not good code, as we shall see shortly, it just illustrates the effect of shadowing a special variable)

```
(let ((a 4))
  (declare (special a))
  (let ((y (lambda (x) (cons x a))))
    (let ((a 'foo)) ; inner not special
      (funcall y 'gloop))))
(GLOOP . 4)
```

The innermost binding of a is not declared special and has no effect on the outer special binding, and thus no effect on its use in y

Answers – Variables

Contrast with this version where the inner binding *is* special:

```
(let ((a 4))
  (declare (special a))
  (let ((y (lambda (x) (cons x a))))
    (let ((a 'foo))
      (declare (special a))
      (funcall y 'gloop))))
(GLOOP . FOO)
```

In EL, the latter returns `(gloop . 4)` – both bindings are lexical as special declarations are ignored and have no effect (EL 13.14)

Answers – Variables

The worse problem with the example is that the lambda contains a reference to a special variable.

```
; SLIME 2.27
CL-USER> (defun dodgy-ref ()
           (let ((a 3))
             (declare (special a))
             (lambda (x) (+ x a))))
DODGY-REF
CL-USER> (let ((y (dodgy-ref)))
   (funcall y 1))
; Evaluation aborted on #<UNBOUND-VARIABLE A {100420B0C3}>.
```

We do the same as before but the lambda doesn't work outside of the `let` – we do not have a closure. The reason is the *dynamic extent* (duration) of the special binding: once the `let` is exited, the binding no longer exists. This is why lexical bindings must have indefinite extent.

Answers – Variables

Contrast this with defining a "global" variable with `defvar`:

```
CL-USER> (defvar *a* 3)
*A*
CL-USER> (let ((*a* 4)) ; not explicitly special
           (+ 1 *a*))
5
CL-USER> (flet ((y (x) (+ x *a*)))
           (let ((*a* 4))
             (y 1)))
5
CL-USER> *a*
3
```

Note the inner binding does not declare `*a*` special but the effect is as if it were declared, as it affects the outer binding of `y`'s use of `*a*`. A (toplevel) `defvar` cannot be shadowed lexically!

Answers – Variables

- In CL, `defparameter` also defines "global" variables (dynamic extent and indefinite scope)
  - We assume `defvar` and `defparameter` are executed at toplevel
- In EL, the only way to declare a special variable is with `defvar`
  - (The reason specialness is pervasive with toplevel `defvar` is `defvar` *proclaims* the symbol special (a proclamation is kind-of a pervasive declaration))
- While the extent is dynamic, a toplevel `defvar` binding effectively remains until Emacs is exited (some people close Emacs...), or the binding is explicitly changed
- Note that if the variable *is already bound*, `defvar` has no effect

```
(defvar znork 'grulp)
znork
(defvar znork 'blazp)
znork
znork
grulp
```

Answers – Functions

Repeating the exercise with functions is now straightforward(ish):

```
CL-USER> (labels ((foo (x) (+ x 1))
                  (bar (k) (* 2 (foo k))))
           (flet ((foo (y) (ash y 4)))
             (bar 3)))
8
```

(This code doesn't work in EL, though it will if you replace the inner `flet` with `labels`; the outer binding is made with `labels` so `bar` can reference `foo`)

The calculation done here is `(* 2 (+ 3 1))`, ignoring the inner binding of `foo`, i.e. the binding is lexical (to be precise, the scope is lexical; the extent indefinite)

Answers – Functions

Compare with this EL (CL is the same):

```
(defun foo (x) (+ x 1))
foo
(labels ((bar (k) (* 2 (foo k)))
         (foo (y) (ash y 4)))
  (bar 3))
96
```

The calculation done now is `(* 2 (ash 3 4))`.

Like `defvar`, (toplevel) `defun` *proclaims* specialness of the binding assigned to its symbol, and the inner `foo` shadows the outer one and the shadowing is visible to `bar` even though `bar` does not "see" the inner definition of `foo` directly

Summa Summarum

- Lexical scope, indefinite extent (“lexical scope”)
  - Variables defined with `let`
  - Functions defined with `flet`, `labels`
- Indefinite scope, dynamic extent (“dynamic scope”)
  - Variables defined with `let` and declared `special` (CL only)
  - Variables defined at toplevel with `defvar` (and CL, `defparameter`)
  - Functions defined at toplevel (or in a toplevel `let`) with `defun`
  - `catch`es
- Lexical scope, dynamic extent
  - Blocks (including implicit blocks) and tags
  - Variables declared `dynamic-extent` (stack allocated, CL only)
- Indefinite scope, indefinite extent
  - Though formally special, constant variables (`most-negative-fixnum`) have indefinite extent and can neither be shadowed nor be unbound
  - `pi` is not special in EL but is constant
  - Symbols naming variables which reference themselves (`nil`, keywords)