Programming Languages & Paradigms
PROP HT 2011

Lecture 14

Functional Programming II – Laziness, Seqs & Macros

Beatrice Åkerblom
beatrice@dsv.su.se

Eager vs. Lazy

- Many programming languages are eager in that arguments to functions are immediately evaluated when passed, and Clojure in most cases follows this pattern as well

```
user=>  (- 13 (+ 2 2))
9
```

- The expression (+ 2 2) is eagerly evaluated, in that its result 4 is passed on to the subtraction function during the actual call, and not at the point of need

Eager vs. Lazy

- In a lazy programming languages, e.g. Haskell, the function argument will be evaluated only if that argument is needed in some computation
  – Laziness can be used to avoid nontermination, unnecessary calculations, and even combinatorially exploding computations

- Familiar example of laziness:

```
if (obj != null && obj.isWhatiz()) {
    ...
}
```


`sequential`, `sequence`, and `seq`

- A `sequential` collection is one that holds a series of values without reordering them
- A `sequence` is a sequential collection that represents a series of values that may or may not exist yet
- `seq` is Clojure's API for navigating collections (`take, nth, drop, interleave, cycle, partition, map, apply, reduce`, ...)

```
user=> ds
[:willie :barnabas :adam]

user=> (first ds)
:willie

user=> (rest ds)
(:barnabas :adam)
```

Everything Is a Sequence

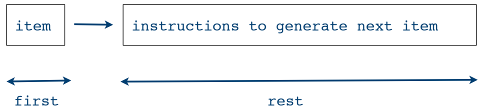
Every aggregate data structure in Clojure can be viewed as a sequence.
A sequence has three core capabilities:

- You can get the first item in a sequence:
  - `(first aseq)`
  - first returns `nil` if its argument is empty or `nil`
- You can get everything but the first item, the rest of a sequence:
  - `(rest aseq)`
  - rest returns an empty seq (not `nil`) if there are no more items.
- You can construct a new sequence by adding an item to the front of an existing sequence. This is called consing:
  - `(cons elem aseq)`

The `seq` function will return a seq on any seq-able collection:
- `(seq coll)`

Lazy Sequences

- The first/rest architecture of the sequence is the basis for laziness
- Lazy sequences is a simple and efficient way to operate on data sets too large to be loaded into the computer's memory at once
- The rest part doesn't necessarily need to exist



Example of Laziness


```
user=> (defn square [x]
    (do
    (println (str "Processing: " x))
    (* x x)))
#'user/square

user=> (map square '(1 2 3 4 5 6 7))
(Processing: 1
Processing: 2
1 Processing: 3
4 Processing: 4
9 Processing: 5
16 Processing: 6
25 Processing: 7
36 49)
```


Example of Laziness, cont'd


```
user=> (def result (map square '(1 2 3 4 5 6 7)))
#'user/result

user=> (nth result 2)
Processing: 1
Processing: 2
Processing: 3
9

user=> (nth result 2)
9

user=> (println result)
(1 4 Processing: 4
9 Processing: 5
16 Processing: 6
25 Processing: 7
36 49)
```

Constructing Lazy Sequences

- To construct a lazy sequence manually in Clojure, the sequence is wrapped in the built-in lazy-seq macro, which handles the magic

```
(lazy-counter 0 2) -> (0 2 4 6 8 10 12 14 16 18 ...)
```


Constructing Lazy Sequences Manually

- To construct a lazy sequence manually in Clojure, the sequence is wrapped in the built-in lazy-seq macro, which handles the magic

```
(lazy-counter 0 2) -> (0 2 4 6 8 10 12 14 16 18 ...)

(defn counter [base increment]
   (cons base (counter (+ base increment) increment)))
```


Constructing Lazy Sequences Manually

- To construct a lazy sequence manually in Clojure, the sequence is wrapped in the built-in lazy-seq macro, which handles the magic

```
(lazy-counter 0 2) -> (0 2 4 6 8 10 12 14 16 18 ...)

(defn counter [base increment]
   (cons base (counter (+ base increment) increment)))


user=> (defn lazy-counter [base increment]
   (lazy-seq
   (cons base (lazy-counter (+ base increment) increment))))
#'user/lazy-counter
```


Constructing Lazy Sequences Manually

- To construct a lazy sequence manually in Clojure, the sequence is wrapped in the built-in lazy-seq macro, which handles the magic

```
user=> (defn lazy-counter [base increment]
   (lazy-seq
   (cons base (lazy-counter (+ base increment) increment))))
#'user/lazy-counter

user=> (take 10 (lazy-counter 0 2))
(0 2 4 6 8 10 12 14 16 18)

user=> (nth (lazy-counter 2 3) 1000000)
3000002

user=> (defn counter [base increment]
   (cons base (counter (+ base increment) increment)))
#'user/counter

user=> (counter 0 2)
StackOverflowError   user/counter (NO_SOURCE_FILE:24)
```

Lazy Sequences Through Sequence Generator Functions

- It's often easier to use sequence generators than the lazy-sec macro directly (`iterate, repeat, range, cycle`, ...)

```
user=> (def integers (iterate inc 0))
#'user/integers

user=> (take 10 integers)
(0 1 2 3 4 5 6 7 8 9)

user=> (defn lazy-counter-iterate [base increment]
   (iterate (fn [n] (+ n increment)) base))
#'user/lazy-counter-iterate

user=> (nth (lazy-counter-iterate 2 3) 1000000)
3000002
```


Lazy Sequences – Losing Your Head

- If you manage to hold onto the head of a sequence somewhere within a function, then that sequence will be prevented from being garbage collected.

```
user=> (let [r (range 1e9)] [(first r) (last r)])
[0 999999999]

user=> (let [r (range 1e9)] [(last r) (first r)])
OutOfMemoryError Java heap space  java.lang.Long.valueOf
(Long.java:557)
```

- Clojure's compiler can deduce that in the first example, the retention of r is no longer needed when the computation of (last r) occurs, and therefore aggressively clears it

Next vs. Rest

- Rest returns a seq, that might be empty or contain elements

```
user=> (rest '(3))
()
```

- Next returns nil if the rest of the seq is empty. This means that we need to look at the rest of the list to determine if it should be a seq or nil

```
user=> (next '(3))
nil
```

- Next is less lazy than rest

Data Structures

Sequence Library

```
user=> (drop 2 '(1 2 3 4 5))
(3 4 5)

user=> (take 10 (cycle (range 3)))
(0 1 2 0 1 2 0 1 2 0)

user=> (interleave [:a :b :c] [1 2 3 4 5])
(:a 1 :b 2 :c 3)

user=> (partition 3 '(1 2 3 4 5 6 7 8 9))
((1 2 3) (4 5 6) (7 8 9))

user=> (map vector[:a :b :c] '(1 2 3))
([:a 1] [:b 2] [:c 3])

user=> (apply str (interpose \, "qwerty"))
"q,w,e,r,t,y"

user=> (reduce + (range 100))
4950
```


Sequence Library

```
user=> (first {:fname "Alonzo" :lname "Church"})
[:lname "Church"]

user=> (rest {:lname "Church" :fname "Alonzo"})
([:fname "Alonzo"])

user=> (cons [:langname "Lambda calculus"] {:lname
"Church" :fname "Alonzo"})
([:langname "Lambda calculus"] [:lname "Church"] [:fname
"Alonzo"])

user=> (first #{:the :quick :brown :fox})
:brown

user=> (rest #{:the :quick :brown :fox})
(:quick :fox :the)

user=> (cons :jumped #{:the :quick :brown :fox})
(:jumped :brown :quick :fox :the)
```

- Maps and sets have a stable traversal order, but that order depends on implementation details, and you should not rely on it

`conj` and `into`

Both conj and into add items at an efficient insertion spot for the underlying data structure

- For lists, conj and into add to the front:

```
user=> (conj '(1 2 3) :a)
(:a 1 2 3)

user=> (into '(1 2 3) '(:a :b :c))
(:c :b :a 1 2 3)
```

- For vectors, conj and into add elements to the back:

```
user=> (conj [1 2 3] :a)
[1 2 3 :a]

user=> (into [1 2 3] [:a :b :c])
[1 2 3 :a :b :c]
```


Macros

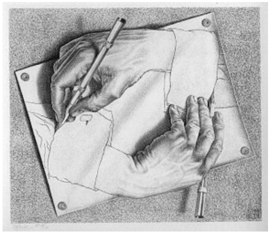
If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.

—Guy Steele

Data is Code is Data

- A Clojure program is made entirely out of data
- Function definitions in Clojure programs are also represented using an aggregation of the various data structures we use to represent data
- Expressions representing the execution of functions and the use of control structures are also data structures

- When a program is the data that composes the program, then you can write programs to write programs



Macros

- A Clojure macro is a construct which can be used to transform or replace code before it is compiled
- Syntactically, a macro is similar to a function, but there are some important differences:
  - A macro shouldn't return a value, but a form
  - Arguments to macros are passed in without being evaluated. They can be altered, ignored or added to the macro's output
  - Macros are evaluated only at compile time

- The macro expression will be replaced with the expression returned by the macro

Macros

- Some are supplied with Clojure
  - `and, or, when, defmacro, defn, lazy-seq, doc` ...
- New ones can be defined by user

Triple-do

- The expression

```
(triple-do (println “Hello”))
```

- Should be compiled to:

```
(do (println “Hello”) (println “Hello”) (println “Hello”))
```

- There’s really no need (except for debugging) for the programmer to see the expansion

```
user=> (triple-do (println “Hello”))
Hello
Hello
Hello
nil
```


Triple-do, cont’d

- `defmacro` is a macro that defines a function and registers it as a macro with the compiler. When the compiler finds the macro, the function will be called and the resulting value will be used to replace the original expression

```
user=> (defmacro triple-do [form]
    (list 'do form form form))
#'user/triple-do

user=> (triple-do (println "does it work?"))
does it work?
does it work?
does it work?
nil
```


Infix Operators

- The expression

```
(infix (1 + 1))
```

- Should be compiled to:

```
(+ 1 1)
```

- Knowing that:

```
user=> (second '(2 + 3)) => +
user=> (first '(2 + 3)) => 2
user=> (nnext '(2 + 3)) => (3)
```

- The macro:

```
(defmacro infix [form]
  (cons (second form) (cons (first form)) (nnext form))))


user=> (infix (2 + 3)) => 5
user=> (infix (2 - 1)) => 1
```


What does the macro do?

```
user=> (macroexpand '(defmacro triple-do [form]
    (list 'do form form form)))
(do (clojure.core/defn triple-do ([&form &env form] (list
(quote do) form form form))) (. (var triple-do) (setMacro))
(var triple-do))

user=> (macroexpand '(triple-do (println "does it work?")))
(do (println "does it work?") (println "does it work?")
(println "does it work?"))

user=> (macroexpand '(infix (2 + 3)))
(+ 2 3)

user=> (macroexpand '(infix (+ 2 3)))
(2 + 3)
```

Code Templating

- Another way of creating macros, than manually creating forms, is code templating
- Makes it possible to enter the return forms as literals, splicing values in where they are wanted
- ` – syntax-quote, which can be unquoted by ~ to insert values into the syntax-quoted expression

```
user=> (defmacro template-triple-do [form]
        `(do ~form ~form ~form))
#'user/template-triple-do

user=> (macroexpand '(template-triple-do (println "yes, it works")))
(do (println "yes, it works") (println "yes, it works") (println
"yes, it works"))
us
```


Code Templating, cont'd

```
user=> (defmacro template-infix [form]
   `(~(second form) ~(first form) ~(nnext form)))
#'user/template-infix

user=> (macroexpand '(template-infix (1 / 2)))
(/ 1 (2))
```

- Splicing unquote

```
user=> (defmacro template-infix [form]
   `(~(second form) ~(first form) ~@(nnext form)))
#'user/template-infix

user=> (macroexpand '(template-infix (1 / 2)))
(/ 1 2)
```


Code Templating, cont'd

```
(+ 5 (* 4 (debug-println(/ 4 3))))
```

- Generated code should look like:

```
(let [result (/ 4 3)]
    (println (str "Value is: " result))
     result)
```

- Macro:

```
user=> (defmacro debug-println [expr]
   `(let [result# ~expr]
        (println (str "Value is: " result#))
        result#))
#'user/debug-println

user=> (macroexpand '(debug-println (/ 4 3)))
(let* [result__167__auto__ (/ 4 3)] (clojure.core/println
(clojure.core/str "Value is: " result__167__auto__))
result__167__auto__)

user=> (+ 5 (* 4 (debug-println(/ 4 3))))
Value is: 4/3
31/3
```


Code Templating, cont'd

```
(rand-expr (println "A") (println "B"))
```

- Generated code should look like:

```
(let [n rand-int 2]
    (if (zero? n) (println "A") (println "B")))
```

- Macro:

```
user=> (defmacro rand-expr [form1 form2]
   `(let [n# (rand-int 2)]
        (if (zero? n#) ~form1 ~form2)))
#'user/rand-expr


user=> (rand-expr (println "A") (println "B"))
B
nil
user=> (rand-expr (println "A") (println "B"))
A
nil
user=> (rand-expr (println "A") (println "B"))
B
nil
```

Macro rules of thumb

- Don't write a macro if a function will do. Reserve macros to provide syntactic abstractions or create binding forms
- Write an example usage
- Expand your example usage by hand
- Use `macroexpand`, `macroexpand-1`, and `clojure.walk/macroexpand-all` liberally to understand how your implementation works
- Experiment at the REPL
- Break complicated macros into smaller functions whenever possible

Six Rules of Clojure FP

- Avoid direct recursion. The JVM cannot optimize recursive calls, and Clojure programs that recurse will blow their stack
- Use recur when you are producing scalar values or small, fixed sequences. Clojure will optimize calls that use an explicit recur
- When producing large or variable-sized sequences, always be lazy. (Do not recur.) Then, your callers can consume just the part of the sequence they actually need
- Be careful not to realize more of a lazy sequence than you need.
- Know the sequence library. You can often write code without using recur or the lazy APIs at all
- Subdivide. Divide even simple-seeming problems into smaller pieces, and you will often find solutions in the sequence library that lead to more general, reusable code

The End

References

- Sebesta, R., "Concepts of Programming Languages"
- Fogus, M. and Houser, C., "The Joy of Clojure", 2011
- Halloway, S., "Programming Clojure", 2009
- VanderHart, L. and Sierra, S., "Practical Clojure", 2010
- clojure.org