Laboratory: Clojure (2)

Summary: We continue our exploration of the Clojure programming language, focusing on lazy lists (which Tate calls infinite sequences) and macros.

Prerequisites: The first Clojure lab. Tate, Section 7.3.

Contents:

- Preparation
  - Exercises
    - Exercise 1: Lazy Lists
    - Exercise 2: What’s Going On?
    - Exercise 3: Fun with Map
    - Exercise 4: Silly Sequences
    - Exercise 5: Checking Computation
    - Exercise 6: Compasses
    - Exercise 7: Argument Lists
    - Exercise 8: The Unless Macro
    - Exercise 9: While Loops

- For Those with Extra Time

Preparation

a. Create a directory for the lab.

b. Open a browser window on Tate’s examples, in case you want to try any of them.

c. Follow Shitanshu’s instructions for starting Clojure.

Exercises

Exercise 1: Lazy Lists

Verify that Tate’s examples from the section entitled “Infinite sequences and take” work the way he suggests.

Exercise 2: What’s Going On?

One of Tate’s lines reads
user=> (->> [:lather :rinse :repeat] (cycle) (drop 2) (take 5))

Traditionally, drop takes two parameters, as does take.

What’s going on here?

**Exercise 3: Fun with Map**

a. What do you expect the result of the following to be?

```clojure
user=> (map + (list 1 2 3) (list 4 5 6))
```

b. Check your answer experimentally.

c. What do you expect the result of the following to be?

```clojure
user=> (map + (cycle [1 2 3 4]) (cycle [1 2]))
```

d. Check your answer experimentally.

**Exercise 4: Silly Sequences**

My MathCounts! students regularly get asked silly questions about exponentiation, like “What is the last digit of $2^{111} + 3^{313}$?”

a. Using the `(time exp)` function, determine how long the “obvious” solution to this problem takes. (The obvious solution is to compute the two values, add them together, and mod by 10.)

b. Students quickly figure out that there’s a pattern to the last digit of powers of 2: 2, 4, 8, 6, 2, 4, 8, 6, .... Determine how long the “take the nth element of the last-digit sequence” strategy takes.

**Exercise 5: Checking Computation**

One of the reasons we use lazy evaluation is to avoid unnecessary computation. For example, if we want the tenth element of a list, and each element of the list involves some computation, there’s usually no reason to do the computation for the first nine elements (at least in a pure system).

Determine experimentally whether Clojure avoids this unnecessary computation.

**Exercise 6: Compasses**

a. Try Tate’s compass example given in the section entiteld “defrecord and protocols”.

b. Rewrite the example to do an eight-point compass (adding NE, NW, SE, and SW to the compass points).
Exercise 7: Argument Lists

As you may recall from your time with Scheme, sometimes you want to write functions that take an arbitrary number of parameters (+ is an obvious example). In Clojure, you can do so by adding an ampersand to the argument list. For example

```clojure
=> (defn arg0 [& args] (first args))
=> (arg0 :a :b :c)
:a
```

a. Verify that this works as advertised.

b. Using this technique, write a function that prints out all of its arguments, prefixed by the argument number. E.g.,

```clojure
=> (printargs :foo :bar :baz)
0: :foo
1: :bar
2: :baz
```

Exercise 8: The Unless Macro

a. Try Tate’s unless examples from the section entitled “Macros”.

b. Add an optional “else” clause to the macro. (This exercise is taken from Tate.)

Exercise 9: While Loops

Write a `(while test body)` macro. (Such a macro seems a bit odd in a side-effect-free language, but Clojure does allow some side effects.

For Those with Extra Time

If you find yourself with extra time, begin the assignment.