CS 331 — Getting Started in Clojure

Version 0.5

Mattox Beckman

Contents

About Clojure

Purpose

The purpose of this document is to help a programmer get up to speed using Clojure. It is not meant to be comprehensive, but to serve as a starting point so that other reference materials will make more sense.

I assume that the reader has programmed before, and thus will be familiar with variables, function calls, loops, and the like. I do not assume that the reader's experience is extensive.

This document is organized by sections. Each section will introduce an aspect of Clojure. Included will be discussion, sample code, and some exercises. There are many code examples. You should read this with a running Clojure environment so you can type them in and play with them yourself. This is the fastest way to learn.

When you are finished with these sections you should be ready to start programming!

History of Clojure

Clojure is a dialect of Lisp. Lisp is best understood as a family of languages. This language family is ancient, one of the oldest languages still in use today. The fact that it is still in use is a testimony to its expressive power and its ability to be modified to the will of the programmer. Today there are many dialects of Lisp in use, some quite new. These include Common Lisp, Scheme, Racket, Guile, and of course, Clojure.

In 1958, John McCarthy created the Lisp programming language at MIT. It was meant to be a theoretical exercise, but then one of his graduate students converted it to assembly language, and thus the first Lisp interpreter came into being.

In 1962, the first Lisp compiler was written in Lisp. A language is usually considered a "toy" language until it can compile itself.

In the early 1970's, Sussman and Steele developed Scheme to study a

programming concept called the *actor*.[1] In 1975, the first paper on SCHEME was published.

In the mid 1990's, Matthias Felleisen founded PLT, a group of researchers interested in SCHEME. They produced their own dialect of SCHEME called, appropriately enough, PLT SCHEME. In 2010, they renamed PLT SCHEME version 5.0 to RACKET.

LISP was very influential for a long time, until the dreaded AI Winter. Researchers began to doubt the ability of symbolic logic to mimic human intelligence, and AI (Artificial Intelligence) became less popular. Because LISP was considered an "AI language," it also became less popular.

In spite of this, people continued to work with LISP, writing production code and creating new dialects. We will ignore more of them except for the one that concerns us now. In 2009, Rich Hickey released CLOJURE publicly. Because it could run on the JVM and have access to all of JAVA's libraries, it generated a lot of excitement, and is quickly finding its way into industry.

[1] Actors are entities that have their own threads and states, and can communicate with other actors. They are an early attempt to understand distributed computing, and are still in use today.

Getting CLOJURE

The very short version is "get Leinengen". Leinengen, or `lein` as the command is called, is a CLOJURE build tool. It can run CLOJURE programs, but it can also manage CLOJURE packages very easily. We will be using this extensively in the course. It can be found at `http://leiningen.org`.

You will definitely want a CLOJURE-aware editor. If you want to get started quickly, the IDE called `LightTable` is very promising. It's a simple `.jar` file, and runs almost everywhere. It can use the `vim` keybindings, for those of you who know what those are. This is one of our recommended environments. Visit `http://www.lighttable.com` to try it.

The `eclipse` package has a plugin called `CounterClockwise` that supports CLOJURE, and I am told that it is good. You can find it at `http://code.google.com/p/counterclockwise`.

There are two editors also worth mentioning. The learning curve on them is much higher, but the benefit is also greater once you are familiar with them. The course web site has some resources for both of these.

The first is a program called `emacs`. There are instructions on the course web site for setting that up. It is a very old editor (the professor started learning it 25 years ago, and it was already considered old then), though still in constant development. It is likely the most common environment for CLOJURE programmers.

The second is another ancient editor called `vim`. It is not as powerful as `emacs`, but the keybindings are optimized for touch-typists. You can edit *very* quickly with `vim`.

For a best-of-all-worlds approach, someone has even written a `vim` emulator in `emacs`. It is called `evil`, and the reader can decide if that is just a coincidence or not. This is what the professor uses and will recommend it to

people who expect to program for many years, and are willing to spend some time at the beginning learning how to use it.

Typographic Conventions

The examples in this manual will be of two types: files and interactive environments. A file listing will look like this:

```
(defn plus [a b]
  (+ a b))
```

Running the program from the interactive environment will look like this.

```
% lein repl
nREPL server started on port 43244
REPL-y 0.2.0
Clojure~1.5.1
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)

user=> (load-file "foo.clj")
#'user/plus
user=> (plus 10 20)
30
```

The `% lein repl` is the Unix prompt and command to start CLOJURE, and the `user=>` symbol is the CLOJURE prompt. Your own prompts may look different depending on how your environment is set up.

Most of the examples in this document will be run from the Unix command line, but you could also use one of the editors we mentioned to take advantage of the graphic user interface.

In order to reduce the amount of space it takes to demonstrate CLOJURE code, many documentation writers use the convention of writing an expression followed by a comment displaying its value. The two forms are inline, like this:

```
(plus 10 20)  ; => 30
```

or on the following line, like this:

```
(plus 10 20)
;; => 30
```

Numerics, Arithmetic, and Function Calls

CLOJURE contains many numeric types. The usual ones are there, such as integers and floats. They look like you would expect. Note that the semicolon is the comment character in CLOJURE.

```
10 ; Long Integers
234.345  ; Floats
```

It also has big integers. These allow as many digits as you have memory to store them.

```
91874123897401923740123948 7 ; Bignums
```

CLOJURE has fractions, though we propbably will not use them much. You can convert them to floats if you want.

```
50/15          ; => 10/3
(float 50/15) ; => 3.3333333333333335
```

Function Calls

In CLOJURE, every computation begins with a parenthesis. This is the most noticeable feature of the language, and perhaps the most important. In non-lisps, a function call might look like `f(x,y)`. In CLOJURE, it looks like

```
(f x y)
```

So, to add two numbers 10 and 20, you would write

```
(+ 10 20)
```

To add 3×3 to 4×4 you would write

```
(+ (* 3 3) (* 4 4))
```

There are a few strengths to this setup. First, precedence is *explicit*. You all know the algebraic rules governing times and plus, but there are many other operators. Second, this allows functions to take a variable number of arguments. For example, you can say `(+ 2 4 8)` to get 14, and you can say `(< 10 x 20)` to check if variable x is between 10 and 20.

Here is some example code

```
(+ 10 20 30)     ; => 60
(- 50 20 1)      ; => 29
(* 8 6 7 5 3 9) ; => 45360
(mod 10 4)       ; => 2
```

Mathematics

CLOJURE uses Java's `Math` library to handle machine math. There is a separate library if you want to use such functions on extended precision numbers. We will not be needing that in this class.

```
Math/E           ; => 2.718281828459045
(Math/sqrt 2)    ; => 1.4142135623730951
(Math/exp 34)    ; => 5.834617425274549E14
Math/PI          ; => 3.141592653589793
```

Exercises

Question 1: Using CLOJURE, determine the sum and the product of 2,4,5,10, and 12.

Question 2: Using CLOJURE, verify the famous identity

$$\sqrt{3^2 + 4^2} = 5.$$

Use functions `(Math/sqrt x)` to compute $\sqrt{x}$ and `Math/pow x 2` to compute x^2.

Question 3: This code is supposed to calculate $3^2 + 4^2$. What's wrong with it?

```
(+ Math/pow(3 2) Math/pow(4 2))
```

Creating and Using Variables

Define

There are two common ways to create variables in CLOJURE. The first is with `def`. The syntax is simple: `(def var exp)` defines a variable named `var` and assigns to it the value given by expression `exp`.

```
(def a 10)
(def b (+ 15 5))
(+ a b)          ; => 30
```

Variables created by `def` persist throughout their scope.

Let

The second way to create variables is with the `let` form. The syntax is `(let [v1 e1 ...] body)`. Unlike `def`, the variables created by `let` are temporary and local. They exist only in the `body` part of the let, and then disappear. You can define more than one variable at a time by adding more pairs.

This session illustrates the interaction between local and global variables.

```
(def a 10)
(def b 20)
(let [a 50] a) ; => 50
a              ; => 10
(let [a 20
      b 40]
 (+ a b))      ; => 60
(+ a b)        ; => 30
```

Variables are defined by let one at a time. Subsequent definitions have access to previous definitions. For those of you familiar with other LISP dialects, this is more like `let*`.

```
(def a 10)
(def b 20)
(let [a 20
      b a]
  (+ a b)) ; => 40
```

Exercises

Question 4: What will this do?

```
(let [x 10
      y 20]
  (/ x y))
```

Question 5: What will be the value of `x` after running the code from question 4? (Yes, this is a trick question!)

Question 6: What will this code print out?

```
(def x 10)
(def y 10)
(let [x 5
      y (+ x 1)
      z (* x y)]
   z)
```

Built-in Data Structures

Sequences

The most common data structure in CLOJURE is actually a family of structures called a *sequence*. A sequence has to be able to give you two things: the first element of the sequence and the rest of the sequence beyond the first element.[2]

We actually do not get to know what a sequence looks like in memory! It could be a list, a vector, or one of several other things. In class we will talk about why this is actually a good thing.

[2] The functions to do this are called `first` and `rest`. Note that `rest` will convert its argument into a sequence!

Lists

Lists are built out of a structure known as a *cons cell*. The name *cons* is short for *construct*. A cons cell contains a data element and a reference to another cons cell, or else to an empty list, which is represented by `nil`.

There are many ways to create lists. One way is to string together cons cells.

```
(def a (cons 10
  (cons 20 (cons 30 nil))))
```

This creates the memory diagram given in figure 1.

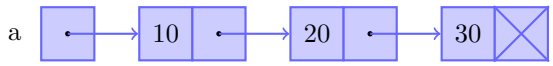

Figure 1: List with Three Elements

Another way to create this is with the quote form. So we could say:

```
(def a '(10 20 30))
```

A third way is to use the `list` form. It is like `cons`, but takes any number of arguments, and evaluates them all.

```
(def a (list 10 (+ 15 5) 30))
a  ; => '(10 20 30)
```

All three of the above codes produce the same memory diagram as given in figure 1.

A list is a kind of sequence, so you can use use `first` to get the first element, and `rest` to access the rest of the elements. The result of `rest` is another sequence, not necessarily a list! Notice also that when you print out a sequence, CLOJURE will use list notation (parentheses) to display it.

```
(first a)  ; => 10
(rest a)   ; => '(20 30)
(rest '()) ; => nil
```

Note that whenever you refer to a cons cell that you have saved in a variable, you do not create a new list. Instead, you recycle the old one.

```
(def x (cons 1 (cons 2 nil)))
(def y (cons 10 x))
(def z (cons 5 z))
```

This code create figure 2.

apply One very nice function is called `apply`. It takes a function and applies it to the whole list. It works with vectors too.

```
(apply + (list 3 5 7 9))  ; => 24
```

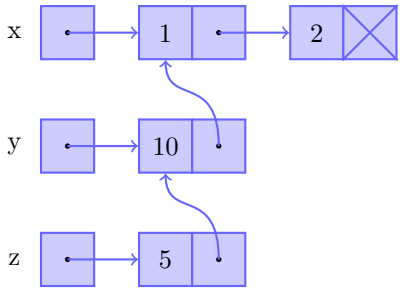

Figure 2: Three Lists Sharing

Common List Operations Here are some list operations you may find useful.[3]

[3] Actually, these are sequence operations. In particular, functions `reverse`, `map`, etc. will return a sequence, not a list. You will probably will not be able to tell the difference most of the time.

```
(def xx (list 2 4 6 8 10))
(count xx)
;; => 5
(reverse xx)
;; => '(10 8 6 4 2)
(append xx xx)
;; => '(2 4 6 8 10 2 4 6 8 10)
(take 2 xx)
;; => '(2 4)
(drop 2 xx)
;; => '(6 8 10)
(first xx)
;; => 2
(rest xx)
;; => '(4 6 8 10)
(second xx)
;; => 4
(nth xx 3)
;; => 8
(map inc '(4 6 8))
;; => '(5 7 9)
```

We have tests:

```
(list? xx) ; => true
(list? 45) ; => false
```

We can also apply functions to the list in different ways.

```
(filter odd? '(2 3 4 5 6)) ; => '(3 5)
(map inc xx)
;; => '(3 5 7 9 11)
(reduce + '(2 3 4 5))      ; => 14
```

```
(reduce * '(2 3 4 5))       ; => 120
(apply max xx)              ; => 10
```

Exercises

Question 7: What is the difference between `v1` and `v2` below?

```
(def a 10)
(def b 20)
(def v1 '(a b))
(def v2 (list a b))
```

Question 8: Consider the following code, an attempt to create figure 1.

```
user=> (def x (cons (cons
      (cons 10 nil) 20) 30))
user=> x
clojure.lang.Compiler CompilerException:
  java.lang.IllegalArgumentException:
  Don't know how to create ISeq from:
  java.lang.Long, compiling ... ; etc.
```

Why did this go wrong?

Question 9: Draw a memory diagram for the following code:

```
(def a (cons 10 (cons 20 nil)))
(def b (cons a a))
(def c (cons b (cons a nil)))
```

Question 10: Try to write some code that creates the diagram in figure 3.

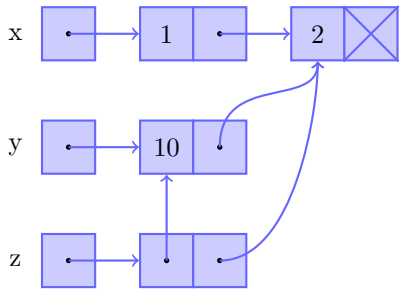

Figure 3: An exercise in memory

Vectors

A *vector*, also known as an *array* in many languages, is a collection of data kept in contiguous memory locations. In CLOJURE, a vector is written with square brackets.

For example this code

```
(def a [10 20 30])
```

creates a memory diagram as in figure 4.

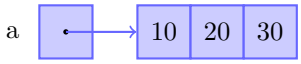

Figure 4: A three-element vector

To access a vector element, you can use the vector as a function that takes an integer argument. That integer will be the index into the vector. The first element always has index 0.

```
(a 0)  ; => 10
(a 2)  ; => 30
```

Vectors in CLOJURE are immutable.

Vectors are also sequences, so you can use `first`, `rest`, `map`, etc. on them as well. You will get a sequence back though. There is a function `mapv` which will return a vector if you really want to preserve the vector type.

Hash Maps

In the second half of the course, you will learn about hash tables in great detail. But they are ubiquitous in CLOJURE code, and very convenient, so we'll talk about how to use them here, and go over the implementation details later.

A hash map is a data structure similar to a vector, but the index can be almost *anything*, not just an integer. The index is usually called a *key*, and the data being indexed is usually called a *value*. The structure as a whole is one of many ways to implement a *dictionary*.

In CLOJURE, we can write a literal hash map using curly braces. Inside the curly braces are pairs. The first element of the pair is the key; the second element is the value.

```
(def ht { "Jenni" "867-5309"
          "Emergency" "911" })
```

One strange fact about CLOJURE is that a comma is considered whitespace. Sometimes you will see hash maps defined this way (note the comma after the first pair) for legibility:

```
(def ht { "Jenni" "867-5309",
          "Emergency" "911" })
```

To look up a value from a hash map, you treat it like a function, with the key as the input.

```
(ht "Jenni") ; => "867-5309"
(ht "Home")  ; => nil
```

Typically, CLOJURE programmers use keywords as the hash map keys when possible. As a result, hash maps tend to look very much like records or structures in other languages. Another advantage of this is that a keyword can be treated as a function that takes a hash map as its argument.

```
(def pt {:x 10 :y 20})
(pt :x) ; => 10
(:y pt) ; => 20
```

To "update" a hash map, we have a function `assoc`.

```
(assoc pt :z 30)
;; => {:z 30, :y 20, :x 10}
(assoc pt :x 123)
;; => {:y 20, :x 123}
```

The keys and values can be arbitrary data structures.

```
(def t { [1 2] "hi"})
t                 ; => {[1 2] "hi"}
(t [1 2])         ; => "hi"
(def np { :p1 {:x 1 :y 2}
          :p2 {:x 5 :y 7}})
(np :p1)          ; => {:y 2, :x 1}
(-> np :p2 :y)    ; => 7
```

The functions `keys` and `vals` will return the keys and values, respectively, of a hash map.

```
(vals pt)   ; => (20 10)
(keys pt)   ; => (:y :x)
```

The Arrow Macro In the previous code, you saw a shortcut, the *arrow macro*. It allows you to rewrite code more compactly.

```
(+  ((np :p2) :y) 10)    ; => 17
(-> np :p2 :y (+ 10))    ; => 17
```

The rule is that the first thing after the arrow becomes the first argument to the second thing. I.e., `(-> x (f a b))` becomes `(f x a b)`. From then on, each result becomes the first argument of the next thing. So `(-> x inc (* 10))` will become `(-> (inc x) (* 10))` and then `(* (inc x) 10))`.

There is a double arrow version, `->>`, in which the results become the last argument. So, `(->> x (f a b))` would become `(f a b x)`.

Because hash maps can be treated like functions, it is common to see the arrow macro used to access values deep within a nested hash map.

Exercises

Question 11: Suppose you had the phone book hash map above and a function `dial` which would actually dial the number. How would you dial all the numbers in the phone book in one line of CLOJURE?

Question 12: The `np` hash map had a bunch of "named points" in it. Suppose we wanted to add the values of the coordinates of `:p2`. Show how to do this.

Creating Functions

There are two common ways to create a function in CLOJURE: the `fn` form and the `defn` form.

Lambda

The keyword `fn` is called `lambda` in most other lisps. That in turn is named for the Greek symbol λ, and represents a function. This notation is from λ-calculus, developed in the 1930's by Alonzo Church to study the dynamics of functions. The λ-calculus is a foundation for the functional languages like CLOJURE, and λ has come to be a near-religious symbol for those who program in these languages. Even CLOJURE uses it in its logo. It was renamed `fn` because we want to be able to use it a lot, and this is much easier to type.

We will not have time to explain in CS 331 why this is so, but will say for now that the ability to define a function and *treat is as any other value* is an extremely powerful mode of expression.

To create a function, use the keyword `fn`, list the parameters you want the function to take, and then write out the body of the function. For example, `(fn [a] (+ a 1))` is a function that takes a variable a and returns the value $a + 1$. If you try to type it out, CLOJUREwill just tell you that it is ... well, it's something.

The CLOJURE logo, with lambda

```
user> (fn [a] (+ a 1))
#<user eval8818 fn__8819 user eval8818 ...
```

To make use of these, you have to apply them. Function application happens just like everything else in CLOJURE: you begin and end with parenthesis.

```
user=> ( (fn [a] (+ a 1))  27 )
28
```

You can define functions that take multiple arguments as well.

```
user=> ( (fn [x y] (* x y))  12 3.5)
42.0
```

Anonymous functions are wonderful, but without names it gets hard to use them more than once. You can assign a function to a variable so you can use it again.

```
user=> (def times (fn [x y] (* x y)))
user=> (times 12 3.5)
42.0
```

Another Shortcut

Programmers are a lazy bunch. Not content to shorten `lambda` to `fn`, we have an even more brief way of writing anonymous functions.

The code `#(+ 2 %)` is equivalent to `(fn [x] (+ 2 x))`. The hash sign in front of the parenthesis says that this expression is a function. The percent symbol is taken as the parameter.

You can create multi-parameter functions by having a number after the percent sign.

```
( #(+ %1 %2) 10 20)  ; => 30
```

This form is commonly used as to supply function arguments to functions like `map` and `reduce`. There is one important limitation though: you cannot nest one of these functions inside another, because the percent argument would be ambiguous.

Exercises

Question 13: Write a function that takes a number and doubles it.

Question 14: Write a function that takes two numbers and returns their hypotneus. Assume you have a function `sqrt` to take the square root.

Defn

Defining functions is so common that we have a shortcut for it. Instead of `(def foo (fn [a] ...))` you can simply say `(defn foo [a] ...)`.

```
(defn double [x] (* x 2))
(defn times [a b] (* a b))
```

Conditionals

To handle branching and conditions, CLOJURE provides several constructs. The two most commonly used are `if` and `cond`.

What is truth?

The `if` form is simple: `(if x y z)`, where `x` is an expression that evaluates to true or false, `y` is the "then" branch, and `z` is the "else" branch. We do not use `then` and `else` keywords like in other languages, the position of the subexpression tells us all we need to know.

You need to know that false is represented as `false` or `nil`. Truth is represented explicitly by `true`, but in fact everything that isn't `false` or `nil` is also considered to be true. In the CLOJURE world, such values are called *truthy*, to distinguish between "things that are taken to be true" and "the boolean value True."

Here is an example function that checks if a number is negative, and if so, makes it positive.

```
(defn quickabs [a]
     (if (< a 0) (- a) a))
```

Sometimes you want an `if` without the else branch. The idiomatic way to do this is CLOJURE is the `when` form. This form is typically used for side-effecting operations, like I/O.

```
(when (< x 0) (println "Negative!"))
;; => "Negative!"
```

You can nest `if` forms if you want to, but if you are doing that, then the next section may be more useful to you.

cond

If you only need two cases, then `if` is usually a good choice, but more frequently you will see the `cond` form, since it allows as many cases as we want. The structure is as follows:

```
(cond c1 e1
      c2 e2
      c3 e3
      ; ...
      :else e4)
```

cooooooond!

The `c` expressions are booleans, and are called *guards*. The corresponding `e` expressions are the branches guarded by the booleans. The first guard in the list that evaluates to true wins, and its corresponding expression gets evaluated.

The final guard is written `:else` here, but any truthy value would do the same thing: catch every case that reaches that point. So, to be clear, the word `:else` has no special meaning. We could have used `:fred` instead, since all keywords are truthy.

Here is a function that finds the maximum of three elements.

```
(defn max3 [a b c]
    (cond (and (> a b) (> a c)) a
          (and (> b a) (> b c)) b
          :else c))
(max3 10 20 3)
;; => 20
```

Question 15: Instead of using `:else` as the last guard, you could also use `true`. Why does that work?

Question 16: The following code is supposed to do a special comparison, but for some reason it is not working. What is the bug?

```
(defn my-compare [a b]
     (cond (= a b) 'equal
           < a b 'less
           :else 'greater))
(my-compare 10 10) ; => equal
(my-compare 10 1)  ; => 10
(my-compare 10 31) ; => 10
```

Records

The `cons` cell is very useful, but sometimes you may want to aggregate your data differently. To teach CLOJURE a new structure, you can use the `defrecord` form. The format is this:

(`defrecord` $name$ $[f_1 \cdots f_n]$)

For $name$ you specify the name of your structure, and for each of the $f_i, 1 \leq i \leq n$ you specify a field name. For those of you with object-oriented backgrounds, the $name$ is like a class name, and f_i are like member variables.

For example, consider this code:

```
(defrecord Foo [x y z])
```

This creates a new structure type called `foo` which has three fields, `x`, `y`, and `z`.

When you create a structure this way, CLOJURE creates a function to create instances of the record (we'll just call it a *constructor*), and accessor functions to retrieve the contents of the fields. The record creator function will have the same name as the record itself, but with a dot appended. This is because `defrecord` creates a Java class, and the name of a class followed by a dot is the name of the constructor for that class. For this reason it is customary to capitalize the record name. The accessor functions will be keywords, having the format : f_i for each of the fields f_i.

In the `Foo` example above, our constructor is called `Foo.` and the field accessors are `:x`, `:y`, and `:z`.

Here is an example of creating a `Foo` and examining the fields.

```
(def f (Foo. 10 20 30))
(:x f)    ; => 10
(:y f)    ; => 20
(:z f)    ; => 30
user> f   ; => #user.Foo{:x 10, :y 20,
                         :z 30}
```

Figure 5 shows what this looks like in memory.

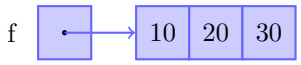

Figure 5: `(def f (Foo. 10 20 30))`

Note that you *cannot* use a record like a function as you could with hash maps. `(f :x)` will raise an error.

Mutation

CLOJURE structures are immutable. To allow destructive update, you have to use special operations. In our class, we will use something called an *atom*.

You create an atom using the `atom` form. To access the contents, you need to use `deref`, or a shortcut . You can set (i.e. destructively update) the atom with a simple value by using `reset!`. It is very common to want to update the content of the atom in terms of its previous content. The `swap!` form allows you to provide a function that does this. So `(reset! a (* @a 2))` becomes `(swap! a #(* % 2))`.

```
(def a (atom 0))    ; => #'user/a
(deref a)           ; => 0
user> @a            ; => 0
(reset! a 5)        ; => 5
(+ @a @a)           ; => 10
user> a             ; => #<Atom@3ebbf380: 5>
(reset! a (inc @a)) ; => 6
(swap! a inc)       ; => 7
```

Looping

You don't need to do it.

You will not use loops as frequently in CLOJURE as you might in other languages. This is because CLOJURE provides constructs to do what you *really*

want to do: in most cases, creating an integer and counting it up is an artificial addition to the problem.

To do something to every element of a data-structure, use `map`.

```
(map inc '(2 3 4 5))    ; => '(3 4 5 6)
```

To select the elements that meet a certain criteria, use `filter`.

```
(filter odd? '(2 3 4 5 6 7 8))
;; => '(3 5 7)
```

To check if every element has a property, use `every?`. To check if some element has a property, use `some`.

```
(every? odd? '(1 2 3 4)) ; => false
(some odd? '(1 2 3 4))   ; => true
```

These functions work on vectors as well as lists.

But I *really* want to use a loop!

Okay. There is a form called `for` that will make a loop for you. It is similar to the `let` form, but instead of a value, you must provide a sequence.

```
(for [i '(1 2 3)] (+ i 10))
;; => (11 12 13)
```

If you are using a loop to do "side-effecting" things, like print values to the string, you will need to wrap the loop in a `dorun` call.

```
user> (for [i '(1 2 3)]
           (printf "%d\n" i))
;; (1
;; 2
;; nil 3
;; nil nil)
user> (dorun (for [i '(1 2 3)]
                  (printf "%d\n" i)))
;; 1
;; 2
;; 3
;; nil
```

The `printf` function takes a format string common to C's, followed by a list of arguments: `%d` means an integer, and `\n` means a newline.

There are some convenient sequence generators for you. One of them is `range`:

```
(dorun (for [i (range 10)]
       (printf "%d-" i)))
0-1-2-3-4-5-6-7-8-9-
(dorun (for [i (range 3 10)]
       (printf "%d-" i)))
3-4-5-6-7-8-9-
```

You can also add *filters*:

```
(dorun (for [i (range 3 10)
             :when (odd? i)]
   (printf "%d-" i)))
3-5-7-9-
(dorun (for [i (range 3 10)
            :when (not (odd? i))]
       (printf "%d-" i)))
4-6-8-
```

You can have multiple sequences. They will be nested.

```
(dorun (for [i (range 0 2)
             j (range 0 2)]
     (printf "%d %d -" i j)))

0 0 -0 1 -1 0 -1 1 -
```

List Comprehensions

The `for` form is also known as a *list comprehension*. Instead of executing a command in the body of the loop, the body emits a value, and `for` accumulates them into a list. You can use the `:when` form to filter out some of the results, and the `:let` form to define temporary variables during the run.

```
(for [i (range 1 10)]
     (* i i))
;; => '(1 4 9 16 25 36 49 64 81)
(for [i (range 1 10)]
     (* i i))
;; => '(1 4 9 16 25 36 49 64 81)
(for [i (range 1 10)
      :when (odd? i)]
      (* i i))
;; => '(1 9 25 49 81)
(for [i (range 1 10)
      :let [j (- i 1)]]
     (* i j))
;; => (0 2 6 12 20 30 42 56 72)
```

Question 17: Write a function `(mods n xx)` that uses `for` to return the moduli of each element of xx with respect to n.

Question 18: Write a function `crazy` that takes two lists `xx` and `yy`. Use a `for` to compute $x^2 - 1 + y^2 - 1$ for odd x^2 and y^2, where $x \in xx$ and $y \in yy$. *Only compute the squares once for each element.* So, `(crazy '(1 2 3) '(4 5 6))` will return `'(26 34)`, because $26 = 1^2 + 5^2$ and $34 = 3^2 + 5^2$.

Namespaces

Often we will write code where some of it should be accessible to the user but other parts should not be. We also will want to group related functions together. In CLOJURE, this is accomplished using *namespaces*.

To declare a module, you use the form

```
(ns myname
   ...)
```

where "myname" is the name of the module you want to define.

Usually, you will use `myname.clj` as the name of your source file.

Let's suppose we want to create a module `foo` that has an increment function. We can write our module like this:

```
user> (ns foo)
nil
foo> (defn bar [x] (+ x 1))
#'foo/bar
foo> (bar 10)
11
```

Notice how the prompt changes.

If you want to change namepaces, use the `in-ns` form. The argument to `in-ns` must be a *symbol*. In the following example we switch namespaces back to `user`.

```
foo> (in-ns 'user)
#<Namespace user>
user> (bar 10)
CompilerException RuntimeException:
  Unable to resolve symbol: bar
  compiling:(NO_SOURCE_PATH:1:1)
```

To use a symbol in a different namespace, you can require the namespace like this.

```
user> (require '[foo :as foo])
nil
```

```
user> (foo/bar 10)
11
```

If you don't want to have to prefix everything with the name of the namespace, you can use this form:

```
user> (require '[foo :refer :all])
nil
user> (bar 10)
11
```

Sometimes you will see `(use 'foo)`; it does the same thing, but it has been deprecated.

Solutions to exercises

Solution 1 The necessary code is:

```
(+ 2 4 5 10 12)    ; => 33
(* 2 4 5 10 12)    ; => 4800
```

Solution 2 The necessary CLOJURE code is:

```
(+ (Math/pow 3 2) (Math/pow 4 2)))
;; => 5.0
```

Solution 3 The `expt` functions should be written as `(Math/pow 3 2)` and `(Math/pow 4 2)`.

Solution 4 The code will create the fraction $10/20$, and display it as $1/2$.

Solution 5 Variable `x` will no longer have a value, since it only existed in the body of the `let`.

Solution 6 30

Solution 7 The list `v1` will be `'(a b)`, while the list `v2` will be `'(10 20)`. The quote operator used to make `v1` causes everything to be quoted, whereas the `list` form evaluates everything and builds the list from the resulting values.

Solution 8 When using `cons` to build lists, you need to put the element in the first part, and the rest of the list in the second part. The code in this question has it reversed.

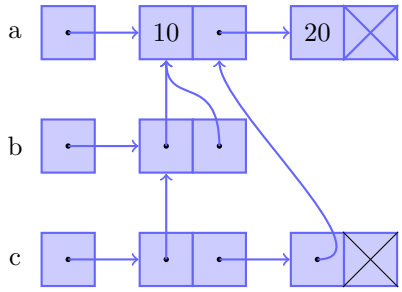

Figure 6: Solution to question 9

Solution 10

```
(def x (cons 1 (cons 2 nil)))
(def y (cons 10 (cdr x)))
(def z (cons y (cdr x)))
```

Solution 11

```
(map dial (vals ht))
```

Solution 12

```
(+ (-> np :p2 :x) (-> np :p2 :y))
```

Solution 13

```
(def double (fn [x] (* x 2))) ; .. or ..
(def double #(* x %)) ; .. or ..
```

Solution 14

```
(def hypot (fn [a b]
   (Maht/sqrt (+ (* a a) (* b b)))))
```

Solution 15 Since it is the last element, it will only be checked if everything else fails, and since it is true, it will always execute if it is reached.

Solution 16 The problem is on this line:

```
        < a b 'less
```

It takes < as a condition, and since it is not false, runs the value a. To fix it, use this code:

```
(defn my-compare [a b]
     (cond (= a b) 'equal
           (< a b) 'less
           :else 'greater))
```

Solution 17

```
(defn mods [n xx]
   (for [i xx]
        (mod i n)))
(mods 3 '(8 6 7 5 3 0 9))
;; => '(2 0 1 2 0 0 0)
```

Usually, though, we would just use `map` to accomplish this.

```
(defn mods [n xx]
   (map #(mod % n) xx))
```

Solution 18

```
(defn crazy [xx yy]
        (for [x xx
              :let [xs (* x x)]
              :when (odd? xs)
              y yy
              :let [ys (* y y)]
              :when (odd? ys)]
             (+ xs ys)))
(crazy '(1 2 3) '(4 5 6))
;; => (26 34)
```

Colophon

This document was compiled using XƎLaTeX and the `tufte-book` package. The body text is set in the *Equity* font, and the headers are set in the *Concourse* font. Both these fonts are available from Matthew Butterick. The source code is set in *Computer Modern Teletype*, designed by Donald Knuth. The picture of Captian Kirk screaming belongs to Paramount Pictures, and is used under the fair use doctrine. The Clojure logo was designed by Tom Hickey (the brother of Clojure's creator, Rich Hickey), and is available under the Creative Commons Attribution-Share Alike license.