# POETRY OF PROGRAMMING

CODE READING EXERCISES IN CLOJURE
V2019.6.13

## Contents


1. Introduction 1
1.1. Why do we need to read code? 1
1.2. Instructions 2
1.3. Recommended method 2
2. Function composition first 2
3. Arithmetic done with functions 3
4. Asking yes-or-no questions: predicates 4
5. Strings 6
6. List, the most fundamental collection 8
7. Vectors 9
8. Making memories 11
9. Defining functions 12
10. Manual function calls 13
11. Lazy lists of numbers 13
12. Functional collection transformation 14
13. Selections 15
14. Conditionals 15
15. Reduce 16
16. Hash-map 17
17. Hash-sets 18
18. Sequence abstraction 19
19. Iteration 19


Sections correspond to chapters of the book https://egri-nagy.github.io/popbook/.

## 1. Introduction

1.1. **Why do we need to read code?** Reading code is executing the program in our head. It is a necessary skill for understanding programs written by others. It is also important for comprehending our own code. When trying to solve a problem, we seldom get it right at first try. So we should ask 'What have I said here?', that is reading the code just written back. Common mistake is to be convinced that we wrote exactly what we wanted.

In order to write programs we need to know the basic building blocks very well. The following exercises all focus on a single aspect of a data structure, a function or a special form. Solving these will provide the background knowledge in the sense of 'knowing the tools'.

These exercises alone will not teach how to write code. One has to solve actual problems for that. Most questions are typical usage examples, but some of them are twisted (if not wicked).

1.2. **Instructions.** *What is the output?*, *What does the expression evaluate to?* The task is to figure out the 'meaning' of the piece of code in the box, which is the result of the computation. Therefore, there is only a single correct answer, the exact output.

- *Each box is a separate question*, definitions and symbol bindings are only valid in the box they appear in.
- *The answer is the value of the last expression/line.* If the box contains definitions, their outputs (which can be system dependent) need not be written.
- When the expression leads to an error message, *it is enough to state the error*, no need to specify which exception is thrown.
- *Indicate the type of the output clearly.* For example, sequence operations return sequences that are lists. When the output is `(1 2 3)`, writing `1 2 3` or `[1 2 3]` instead are not acceptable answers. Similarly, when the output is `"hello"`, then writing `hello` is incorrect. When the answer is `true`, writing yes will not earn a mark.

1.3. **Recommended method.** First, solve these problems without the computer. Write down the answers, then check them in a REPL by evaluating the code snippets. If your answer is different, then check your notes or textbook or documentation for the given function and data structure. Simply copy-pasting the expressions into the REPL without thinking achieves very little learning. This is also the reason for not giving the solutions in this document.

Here are some hints for building up reading skills. It makes sense to keep asking these questions.

- Data literals (e.g. numbers, strings, characters) are easy to read. They just evaluate to themselves. Their meaning is immediate.
- For a symbol, we need to find what does it bind to? What is the meaning? Is it a function? Is it a data item? We need to go backwards to find where the symbol gets defined. Is it in a `let` statement? Is it defined in the environment? Is it an argument to a function?
- If the expression is composite, work from the inside out. Evaluate the sub-expressions first. Make notes. Rewrite the full expression by substituting the values for the sub-expressions.
- If a function is called, what are its input arguments? Does the function expect a number of arguments? or a single collection?

## 2. Function composition first

```
10
```

⟶

```
(identity 10)
```

⟶

```
(inc 2)
```

⟶

```
(dec 9)
```

⟶

```
(inc (dec 0))
```

⟶

```
((comp inc dec) 0)
```

⟶

```
(dec (dec 0))
```

⟶

```
((comp dec dec dec) 0)
```

⟶

```
((comp (comp dec dec) (comp dec dec)) 5)
```

⟶

```
((comp identity identity) 1)
```

⟶

```
((comp identity inc) 1)
```

⟶

```
(inc 11.11)
```

⟶

```
(dec 0.01)
```

⟶

```
((comp inc) 1)
```

⟶

```
((comp) 42)
```

⟶

```
(((comp comp comp) dec dec) 1)
```

⟶

## 3. Arithmetic done with functions

```
(+ (- 6 4) (/ 6 3))
```

⟶

```
(+ 1 (* 2 (- 1 (/ 12 4))))
```

⟶

```
(+)
```

⟶

```
(-)
```

⟶

```
(*)
```

⟶

```
(/)
```

⟶

```
(+ 3)
```

⟶

```
(* 5)
```

⟶

```
(/ 3)
```

⟶

```
(- 5)
```

⟶

```
(+ (* 5 2) (- 4 3) (*))
```

⟶

```
((comp - *) 2 3 4)
```

⟶

```
((comp / *) 2 3)
```

⟶

```
(* (inc 1) (inc (inc 1)))
```

⟶

```
(/ 10 4)
```

⟶

```
(/ 10.0 4)
```

⟶

```
(/ 10 4.0)
```

⟶

## 4. Asking yes-or-no questions: predicates

```
(zero? 0)
```

⟶

```
(zero? 1)
```

⟶

```
(pos? 1)
```

⟶

```
(pos? 1111)
```

⟶

```
(pos? 0)
```

⟶

```
(neg? 0)
```

⟶

```
(neg? -2)
```

⟶

```
(= (+ 1 2 3) (* 1 2 3))
```

⟶

```
(<= 2 2 2)
```

⟶

```
(<= 2 1 2)
```

⟶

```
(< 2 1 2)
```

⟶

```
(< 2 2 2)
```

⟶

```
(< 2 3 4)
```

⟶

```
(> 2 3)
```

⟶

```
(> 2 2)
```

⟶

```
(>= 2 2)
```

⟶

```
(fn? +)
```

⟶

```
(fn? -)
```

⟶

```
(fn? identity)
```

⟶

```
(fn? (+))
```

⟶

```
(number? (+))
```

⟶

```
(rational? (/ 7 3))
```
⟶
```
(rational? 2)
```
⟶
```
(float? 2)
```
⟶
```
(float? 2.0)
```
⟶
```
(integer? (+ 1 2 3 4 5))
```
⟶
```
(integer? (+ 1 2 3.0 4 5))
```
⟶
```
(float? (+ 1 2 3.0 4 5))
```
⟶
```
(number? 12.1)
```
⟶
```
(number? 0)
```
⟶
```
(number? (/ 1 19))
```
⟶

## 5. Strings

```
(char? \x)
```
⟶
```
(char? \space)
```
⟶
```
(char? \8)
```
⟶
```
(= 9 \9)
```
⟶
```
(string? "Granny Weatherwax")
```
⟶
```
(string? "      ")
```
⟶

```
(string? "")
```
⟶
```
(string? "12")
```
⟶
```
(number? "12")
```
⟶
```
(= \space " ")
```
⟶
```
(= (str \space) " ")
```
⟶
```
(str \1 2 "3" (- 5 1))
```
⟶
```
(str "The answer:" 42)
```
⟶
```
(str "The answer: " 42)
```
⟶
```
(str "The answer: " 42 ".")
```
⟶
```
(str "The answer: " (* 6 7) ".")
```
⟶

WARNING! The following questions assume that the string library is loaded by `(require '[clojure.string :as string])`.

```
(string/upper-case "helloooo!")
```
⟶
```
(string/capitalize (string/lower-case "HELLO!"))
```
⟶
```
(string/ends-with? "mango" "go")
```
⟶
```
(string/ends-with? "mango" "GO")
```
⟶
```
(string/starts-with? "How to solve it?" "?")
```
⟶
```
(string/replace "banana" "a" "e")
```
⟶

```
(string/replace (string/replace "banana" "a" "-") "-n" "x")
```

⟶

## 6. List, the most fundamental collection

```
'(1 2 3)
```

⟶

```
(1 2 3)
```

⟶

```
(list 4 5)
```

⟶

```
(list 4 5 '(6 7))
```

⟶

```
(list 4 5 '(6 7 (8 9)))
```

⟶

```
(cons 11 '(19 13))
```

⟶

```
(cons 7 (cons 5 (list 3 4)))
```

⟶

```
(first '(8 3 5))
```

⟶

```
(first '())
```

⟶

```
(rest (5 3 8))
```

⟶

```
(rest '(5 3 8))
```

⟶

```
(rest '())
```

⟶

```
((comp rest rest) '(2 3 4))
```

⟶

```
((comp first rest) '(2 3 4))
```

⟶

```
((comp first first) '(2 3 4))
```

```
⟶
(last '(9 3 2))
⟶
(last '())
⟶
(reverse '(1 2 3))
⟶
((comp first reverse) '(1 2 3))
⟶
(empty? ())
⟶
(empty? '(1))
⟶
(count ())
⟶
(count (list \a \b))
⟶
(concat '(1) '(3) '(2))
⟶
(concat)
⟶
```

## 7. Vectors

```
(vector \a \b \c)
⟶
(vector [1 2])
⟶
(vec [1 2])
⟶
(vec '(\a \b))
⟶
(vec [])
⟶
```

```
(vector [])
```

⟶

```
(count [1 "two" 3])
```

⟶

```
(count [])
```

⟶

```
(count [[]])
```

⟶

```
(count [[[]]])
```

⟶

```
(count [[[[[[[[]]]]]]])
```

⟶

```
(count [[][][]])
```

⟶

```
(nth [\a \b \c] 0)
```

⟶

```
(nth [\a \b \c] 2)
```

⟶

```
(nth [\a \b \c] 3)
```

⟶

```
([\a \b \c] 0)
```

⟶

```
([\a \b \c] 1)
```

⟶

```
([\a \b \c] 3)
```

⟶

```
(nth [] 0)
```

⟶

```
(conj [] 1)
```

⟶

```
(conj [] 1 2)
```

⟶

```
(conj [] 1 2 3)
```

⟶

```
(conj [\a \b]  3)
```

⟶

```
(conj [] [])
```

⟶

```
(conj [] [] [])
```

⟶

```
(conj [1 2] [3])
```

⟶

```
(vec (range 3))
```

⟶

```
(vector (range 3))
```

⟶

```
((comp ["hello" "world" "!"] [2 1 0]) 2)
```

⟶

## 8. Making memories

```
(def a 2)
(* 3 a)
```

⟶

```
(def a 5)
(def b 7)
(* a b)
```

⟶

```
(= 'x "x")
```

⟶

```
(def x "x")
(= "x" x)
```

⟶

```
(def x "grape")
(def x "apple")
x
```

⟶

```
(let [x 2] (* 3 x))
```

⟶

```
(let [x 2, y 7] (* y x))
```

⟶

```
(let [s "world"] (str "Hello " s "!"))
```

⟶

```
(let [s "everyone"] "Hello world!")
```

⟶

```
(def x 2)
(let [x 100] (inc x))
```

⟶

```
(def x 2)
(let [x 100] (inc x))
x
```

⟶

```
(let [x 0 y (inc x) z (dec y)] [x y z])
```

⟶

## 9. Defining functions

```
((fn [x] (+ x 2)) 0)
```

⟶

```
((fn [x] (+ x 2)) 1)
```

⟶

```
((fn [x y] (+ x (* 2 y))) 1 2)
```

⟶

```
((fn [x] (+ x (* 2 y))) 1 2)
```

⟶

```
(def x 31)
((fn [x] (+ x 2)) 1)
```

⟶

```
(defn f [s] (str "Input was " s "."))
(f 42)
```

⟶

```
(defn f [n] (* 3 n))
((comp inc f) 1)
```

⟶

## 10. Manual function calls

```
(+ [1 2 3])
```

⟶

```
(apply + [1 2 3])
```

⟶

```
(max 5 2 7 4)
```

⟶

```
(max [5 2 7 4])
```

⟶

```
(apply max [5 2 7 4])
```

⟶

```
(apply str [\h \i])
```

⟶

## 11. Lazy lists of numbers

```
(range 2)
```

⟶

```
(range 1 2)
```

⟶

```
(range 1 2 3)
```

⟶

```
(range 2 1 3)
```

⟶

```
(range 0 1 0.4)
```

⟶

```
(take 2 (drop 2 (range)))
```

⟶

```
(take 3 '(3 3 3 3))
```

⟶

```
(take-while neg? (range -2 2))
```

⟶

```
(drop-while neg? (range -2 2))
```

⟶

```
(take-while zero? '(-1 0 1))
```

⟶

```
(drop-while zero? '(-1 0 1))
```

⟶

```
(drop-while zero? '(-1 0 1 -1))
```

⟶

```
(drop-while neg? '(-1 0 1 -1))
```

⟶

```
(count (take 5 (range)))
```

⟶

## 12. Functional collection transformation

```
(map inc [1 2 3 4 5])
```

⟶

```
(map dec [1 2 3 4 5])
```

⟶

```
(map range [2 3])
```

⟶

```
(mapcat range [2 3])
```

⟶

```
(map str [1 2 3])
```

⟶

```
(map reverse [[1 2] [3 4]])
```

⟶

```
(map reverse (map range [1 2 3]))
```

⟶

```
(map even? (range 5))
```

⟶

```
(map odd? [0 1 2 3])
```

⟶

```
(map str [1 \a "2" ()])
```

⟶

## 13. Selections

```
(filter even? [10 11 12 13])
```

⟶

```
(filter odd? [10 11 12 13])
```

⟶

```
(filter even? [1 3 5 7])
```

⟶

```
(filter number? [1 "2" \3])
```

⟶

```
(filter string? [1 "2" \3])
```

⟶

```
(filter char? [1 "2" \3])
```

⟶

```
(remove nil? [[] nil () 0])
```

⟶

```
(remove even? (range 6))
```

⟶

```
(filter char? [1 \a 2 \b])
```

⟶

## 14. Conditionals

```
(if true 42 24)
```

⟶

```
(if false 42 24)
```

⟶

```
(let [x 2] (cond (string? x) "just a word"
                 (= x 2) "two"
                 true "not two"))
```

⟶

```
(let [x 4] (cond (string? x) "just a word"
                 (= x 2) "two"
                 true "not two"))
```

⟶

```
(let [x "2"] (cond (string? x) "just a word"
                   (= x 2) "two"
                   true "not two"))
```

⟶

```
(not nil)
```

⟶

```
(not 42)
```

⟶

```
(and false nil)
```

⟶

```
(or false nil)
```

⟶

```
(and 1 2)
```

⟶

```
(or 1 2)
```

⟶

## 15. Reduce

```
(reduce conj [9] [2 3] )
```

⟶

```
(reductions conj [9] [2 3] )
```

⟶

```
(into [9] [2 3] )
```

⟶

```
(into '(1 2) '(3 4 5))
```

⟶

```
(reductions conj '(1 2) '(3 4 5))
```

⟶

```
(reduce max 0 [2 5 3])
```

⟶

```
(reduce max 6 [2 5 3])
```

⟶

```
(reductions max 0 [2 5 3])
```

⟶

```
(reductions max 6 [2 5 3])
```

⟶

```
(defn rf
  [v x]
  (if (odd? x)
    (conj v x)
    (conj v (dec x))))
(reduce rf [] [1 2 3 4 5])
```

⟶

```
(defn rf [v x] (if (even? x) (conj v x) (conj v (inc x))))
(reduce rf [] [1 2 3 4 5])
```

⟶

```
(reduce * [1 2 3])
```

⟶

```
(reduce + [1 3])
```

⟶

## 16. Hash-map

```
(hash-map \a \b \b \a)
```

⟶

```
(hash-map :x 10 :y 11)
```

⟶

```
({1 2 3 4} 3)
```

⟶

```
({1 2 3 4} 2)
```

⟶

```
(zipmap (range 4) (reverse (range 4)))
```

⟶

```
(hash-map :a 1 :b 2 :c)
```

⟶

```
(:a {:a "1" :b 2})
```

⟶

```
(1 {1 2 3 4})
```

⟶

```
({:a 1 :b 2} :c)
```

⟶

```
({:a 1 :b 2} :c :nothing)
```

⟶

```
(get {:a 1 :b 2} :c)
```

⟶

```
(get {:a 1 :b 2} :c "No such key!")
```

⟶

```
(map {1 "one" 2 "two"} [0 1 2 3])
```

⟶

```
(assoc {:a 3} :b 2)
```

⟶

```
(keys {:title "Solaris" :author "Stanislaw Lem" :year 1961})
```

⟶

```
(vals {:title "Solaris" :author "Stanislaw Lem" :year 1961})
```

⟶

## 17. Hash-sets

```
(hash-set "hello" 17 \c)
```

⟶

```
(set [81 72])
```

⟶

```
(hash-set [6 28 496])
```

⟶

```
(apply hash-set [6 28 496])
```

⟶

```
(set [6 28 496])
```

⟶

```
(sorted-set "hellooooo!!")
```

⟶

```
(apply sorted-set "hellooooo")
```

⟶

## 18. Sequence abstraction

```
(seq "coffee")
```

⟶

```
(seq (zipmap (range 4) [\a \b \c \d]))
```

⟶

```
(seq {:a 3 :b 2})
```

⟶

```
(seq [1 2 3 4])
```

⟶

```
(count {:a 3 :b 4})
```

⟶

## 19. Iteration

```
(take 4 (iterate inc 2))
```

⟶

```
(take 4 (iterate dec 0))
```

⟶

```
(take 8 (iterate (fn [x] (* 2 x)) 1))
```

⟶

```
(take 4 (iterate (fn [v] (conj v 1)) []))
```

⟶