Attila Egri-Nagy

Poetry of Programming

v2017.07.02

These notes are being written for the MAT245 course at Akita International University in Japan. The content is not complete and stable yet, but each version is meant to be consistent and thus usable for self-study. Comments are welcome, when reporting errors please specify the version number.

As science and technology advance, their fundamental ideas get simpler and better explained. Similarly, computer programming has also become more accessible. Being a software engineer is still as back-breaking as it ever was, but now the thrills of problem solving by instructing computers can be experienced with a little effort.

“THE PROCESS of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.” – the first sentence of the *Art of Computer Programming* by Donald E. Knuth [4].

Here we aim to introduce the ideas of programming in a pure and minimal way. We focus more on conversations with a computer and less on the tools required for software engineering. Hence the language choice: the functional core of Clojure in its interactive command line REPL (read-eval-print-loop). Clojure is from the Lisp family of programming languages. It is an ideal first language with a “cheeky” character: it realizes many programming concepts with ridiculously simple constructs.

Contents

Introduction

What is programming?

How to instruct computers to do something? How to ask them questions? Computers solve problems in an extremely precise manner, therefore any tasks given to them should also be specified with thorough exactness. Work done by computers has a simple general form: based on input information we would like get the desired output. This observation makes our question more specific. *What is the most rigorous way of defining input-output pairs?* The mathematical language of *functions* is the primary form of talking to computers in a programming situation. One way or the other, all programming languages are about using, defining, transforming and composing functions. We specify a function and give its argument values, then the computer figures out the value of the function. We say that we *call* a function and the computer *evaluates* the function with the given arguments. This is how the game called programming is played.

"A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric programming languages that prescribe the tasks we want our processes to perform." – the auspicious beginning of legendary MIT programming textbook, colloquially known as SICP [1].

Programming is also about writing text. Text interpreted and executed by computers, and – an often neglected aspect – text read by human beings. So written code is also a way of communication between humans. Natural language is used not just for everyday communication, but for expressing personality, emotions and beauty. Similarly, beyond the mundane tasks of application development and maintenance, programming languages can express ideas, ingenuity, wisdom, wit and beauty. Here we aim to develop the ability of appreciating the beauty in written code. Slight difference from poetry is that reading text is not enough. The joy of coding can only be experienced by doing it.

There are three levels of understanding of procedural knowledge, i.e. knowing how to do things.

1. Seeing someone else doing it, listening to an explanation.

2. Doing it.

3. Explaining it to someone else.

Replace 'someone else' by 'computer' in the last item, and you will see why writing computer programs is an efficient way of understanding the world around us. Writing code for problem solving requires the ability of pulling things apart and rebuild them from primitive building blocks.

Learning a programming language is like learning a foreign language, but easier and faster than that. The 'grammar', the syntax is less complex, and 'meaning', the semantics, is aimed to be unambiguous. So learning a programming language is easy, while learning a natural language is more difficult. Exactly the opposite what people would think.

The purpose of this essay formed tutorial is to give a lightning-fast introduction to the core Clojure language. By reading and rereading this text and by trying out the examples one should gain enough knowledge to tackle programming puzzles in the style of https://4clojure.com. The imagined reader is someone who may not become a professional software developer, but do not want to miss out on this intellectual adventure (for instance, modern Liberal Arts students). People with some programming experience may find the exposition peculiar if not revolting. The code examples are chosen for maximizing the speed of worldview expansion. This may not coincide with the best practices and accepted idioms for writing efficient and readable code. It is assumed, that the reader will acquire good taste by writing and reading code, and not by merely reading a tutorial text.

http://clojure.org/

What is the purpose, benefit of learning programming? Beyond the oft-repeated usefulness of programming due to the pervasiveness of computing, *learning to code will teach you how to think clearly and efficiently*. And this will be helpful in whatever profession/field you will be working in.

WARNING!!! This text is dense. It introduces at least one new concept in each example. Reading without trying out the examples will have little effect. Copy-pasting is also discouraged, since it does not really contribute to the learning process. **Play in the REPL!**

Function composition first

Programming computers can be done by using functions. We assume at least some vague memories of functions from school, like $f(x) = 4x^2 + 2x - 4$. A function takes a value and produces another value. In this example the input value is a number, for example $x = 1$, in which case f produces the output $f(1) = 2$, since $4 \cdot 1^2 + 2 \cdot 1 - 4 = 4 + 2 - 4 = 2$. Almost all examples of functions in high school math take numbers and produce numbers. This is a very limited usage of the function concept. In general, the input of a function can be anything, text or some composite piece of information; and the same is true for the output.

What is a function in general? Metaphorically, a function is a machine that can produce outputs from input values. We expect that a machine produces *valid outputs for all meaningful input values*. Thus, the sets of possible inputs and outputs are part of the definition of a function. We also expect that we get the *same output for a particular input all the time* (with the notable exception of probabilistic functions).

The mathematical definition of a function is just the precise description of the machine metaphor in the language of set theory and mathematical logic. Also, the set of inputs X is called the *domain*, while the set of outputs is called the *codomain*.

A function $f : X \to Y$ is a subset of the direct product (ordered pairs)

$$X \times Y = \{(x, y) \mid x \in X, y \in Y\}$$

such that

1. $\forall x \in X, \exists y \in Y$: $(x, y) \in f$,
2. $\forall x \in X, \forall y_1, y_2 \in Y$:

$$(x, y_1) \in f \text{ and } (x, y_2) \in f \implies y_1 = y_2,$$

i.e. for each $x \in X$, there is at least and at most one, therefore exactly one ordered pair (x, y) in f.

It is one thing that we can ask a computer to evaluate a function already available, and it is another thing one to produce a function which is needed for our purpose. This is the difference between being a *user* and a *programmer*. There are basically two ways of constructing new functions:

1. Creating new functions by composing existing ones.

2. Writing new functions.

Our first goal is to understand function composition as quickly as possible. Composition is in the sense of putting LEGO bricks together.

List notation for functions

For the sake of precision, we need to develop a new notation for functions, which is more suitable for computers. For a function $f(x)$, the input variable is x is also called the *argument* of the function. For the computer we write `(f x)` instead of $f(x)$. The function symbol

f jumps behind the parenthesis but keeps a bit of distance from its argument. There can be more than one arguments, for example `(g x y z)` has three arguments. Even there are functions with no arguments, simply written as `(h)`. All that matters that functions are written as a *list* denoted by parentheses. The first element is the function, the following ones are arguments. By convention, we also call this list of a function and its arguments a *function call*. One can think of this as grammar rule, a minimal syntax: **functions and its input arguments are written as a list of symbols between opening and closing parens, i.e. round brackets '`(`' and '`)`'.**

In algebra we also write $y = f(x)$. So where is y? That is the computer's answer. The process of coming up with an answer is *function evaluation*. We can think of the computer as a machine that has a penchant for computing functions. An opening paren triggers this habit, and the first symbol, the name of the function, determines what to do with the rest of the list. **The first element of a list is expected to be the name of a function, the rest of the list contains input data items for the function. The default behaviour is to compute the function.** So, in order to get answers for computational problems we have to construct functions whose values are the solutions. This is the essence of functional programming.

The simplest function: `identity`

What is the simplest function? The function that does nothing, $f(x) = x$. Given the input it just returns it back. In elementary mathematics we rarely talk about more than three functions, so the symbols f, g, h are often enough. In programming we define tons of functions, so naming them is quite an issue (some say the biggest). It is thus important to give functions nice names. We call the simplest function `identity`.

These are snippets of conversations with the computer. You enter the function call (the first line), then the answer appears below. Instead of writing an essay or book, which will be read by someone else later, we have this interactive, question-answer style communication.

```
(identity 42)
42
```

It works for numbers as expected. Same for text, but we have to put the sequence of letters and other symbols into double quotes. We will use a more technical term for textual data items, they are *strings*. Numbers and strings are examples of *data literals*. They are values that we write explicitly into the code or the REPL, as opposed to computing them. In other words, they evaluate to themselves. Data literals are literally just data.

```
(identity "Hello World!")
"Hello World!"
```

We give the function a traditional geek greeting in a string and it simply gives it back. No surprise, no excitement. So let's do something unusual.

Other computers may give slightly different answer, as this internal representation of functions is system-dependent.

```
(identity identity)
#<core$identity clojure.core$identity@1804686d>
```

Whoa! What's that? We asked for trouble, and we got it. Something scary came up from deep inside the system. It looks like some sort of internal representation of the `identity` function. We revealed the true identity of `identity`!

Poking the guts of the system is not our purpose here, so we will not pursue this investigation any further. However, what happened offers a remarkable insight. **Functions can take functions as arguments.** While `identity` does this only as a contrived case, we will see that there are functions designed to take other functions as arguments and return functions as well. This is a big deal, this is what makes functional programming functional.

Growing and shrinking numbers: `inc` `dec`

Let's pretend we don't have the numbers (except zero), so we need to construct them. In mathematics, in order to build the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, we need only two things: zero and the function $f(x) = x + 1$. Thus, $1 = f(0)$, $2 = f(f(0))$, $3 = f(f(f(0)))$, and so on. In CLOJURE this function is called `inc`, as an abbreviation for incrementation.

Let's say, we have a faulty keyboard. The digit keys do not work, except zero. Therefore, we do not have a way to enter numbers. Not a realistic scenario, but in foundational investigations in mathematics it is a very important tool (see Peano axioms).

```
(inc 0)
1
```

How can we do bigger numbers? Just like in math, nesting the function, using our slightly different notation.

'Nesting' in computing means that some object can contain some other object of the same kind. In other words, information is represented in a hierarchical manner.

```
(inc (inc (inc 0)))
3
```

Observe, that when a value is expected, we can simply call another function, since functions produce values. So in the general scheme of `(f x)`, the argument `x` can be replaced by something like `(g y)`. For `(f (g y))` CLOJURE will evaluate `(g y)` first, then the value will be fed into `f`. **Function calls can be nested.** By the way, we can also construct negative numbers by `dec`.

```
(dec 0)
-1
```

And no worries, we have all the numbers in CLOJURE.

Composition: `comp`

Nesting function calls has the unwanted effect of parentheses piling up. We can spare them by a simple trick. Instead of nesting the function calls, we can combine the functions first then apply the composite function to the argument. In mathematics we write

$$f \circ g(x) = f(g(x)),$$

while in Clojure
`((comp f g) x)`
means
`(f (g x))`.

Note the order, the rightmost function in the composition gets executed first.

We can compose arbitrary number of functions.

```
((comp inc inc inc inc) 0)
4
```

If $f(x) = x + 1$, then we would write this with function composition $\circ$ in algebra as $(f \circ f \circ f \circ f)(0)$ instead of $f(f(f(f(0))))$. The calculated value is the same, but the two solutions are different, in a – let's say – ontological, metaphysical sense. In the composition case a new entity, a new function is created.

We can compose different functions, if the output of one function can be eaten by the other function. This is true for `inc` and `dec` since they expect and produce numbers.

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

Here we create a function that takes a number, decrements it, then increments the result. This is of course just a very roundabout way of saying that we want the identity function for numbers.

The general identity function can also be produced by `comp`. Given no arguments, it exactly returns that.

The importance of the identity function becomes obvious on the level of abstract algebra: it is the neutral element of functions under composition. What this means for everyday programming is that we have sensible defaults. While most of the time we want to compose two or more functions, nothing bad happens if we try that with less.

```
((comp) 19)
19
```

Consequently, `comp` works for a single argument as well. Given a function, it composes the function with the identity function, so `(comp inc)` is very much like `inc`.

Arithmetic done with functions

Computers, before they did anything else, performed arithmetic calculations. We do not usually think about arithmetic calculations in terms of applying functions, we just simply add, subtract, multiply and divide numbers. This is an inconsistency on our side. It is better to get rid of it, so we will write arithmetic calculations as function applications.

The basic operations: `+ - * /`

Addition is the simplest algebraic operation. Here it is in Lispy style.

```
(+ 2 3)
5
```

Very unusual after writing $2 + 3$ for many years, but there are numerous benefits. First of all, when adding more than two numbers together, we do not need to repeat the `+` symbol.

```
(+ 1 2 3 4)
10
```

A function can have different number of arguments, from zero to many. Surprisingly, we can do addition and multiplication with less than two arguments.

```
(+ 17)
17
(+)
0
(*)
1
```

When there is no argument, `+` and `*` are constant functions. The default values they give come from abstract algebra (additive and multiplicative identities, neutral elements). For subtraction and division both `(-)` and `(/)` complain about not having arguments. But their one argument versions do rely on the default values.

```
(- 1)
-1
(/ 2)
1/2
```

Yes, Clojure can do rational numbers (but ClojureScript has a weaker host, so it cannot).

Being a hosted language means that inner workings of the language are written in a different language. For Clojure the host is Java, for ClojureScript it is JavaScript. This also explains why the error messages look so strange (or familiar): they come from the underlying host.

Another benefit of the function notation is that there is no need for precedence rules. What is the value of $2 + 3 \cdot 4$? Well, it depends. There are two possibilities $(2 + 3) \cdot 4 = 20$, and $2 + (3 \cdot 4) = 14$. So we have to put the parens or agree that multiplication has to be done first. In Lisp the problem is non-existent.

```
(+ 2 (* 3 4))
14
(* 4 (+ 2 3))
20
```

The price to pay: unusual notation; the gain: no ambiguity.

We can also use numbers with decimal fractions, but they are 'contagious'. Once they appear in an arithmetic expression, the result turns into a decimal number as well.

```
(* 2 (+ 1 2 3.45))
12.9
(/ 0.7 2)
0.35
```

Asking yes-or-no questions: predicates

The simplest type of questions are where the answer is yes or no. These are called *predicates* in mathematics, and the computer says `true` and `false` instead of yes and no. It is a nice habit to name predicates ending with a question mark.

```
(zero? 0)
true
(zero? 1)
false
(neg? -1)
true
(pos? -1)
false
```

Some predicates are so obvious that they don't even need a question mark to signal their nature. We could say `smaller?` but we got used to < in mathematics, and the latter is lot easier to type.

It's not exactly a top achievement that the computer can tell that zero is smaller than one. This is just for demonstrating the way of asking the questions. The real usage of predicates will be clear when will use symbols that can mean a wide range of values. Much like in math, $3 < 4$ is a fact, but $x < 4$ is an expression that depends on x, and divides the set of real numbers into two sets (solutions and non-solutions).

```
(< 0 1)
true
(= 1 1)
true
(= 0 1)
false
```

We can check the equality of more than two things in one go.

```
(= 1 (dec 2) (inc 0) (/ 7 7) (+ 3 -2))
true
```

Similarly, we can conveniently check whether the arguments are strictly increasing or non-decreasing.

```
(< 1 2 3 4 5 6)
true
(< 1 2 2 3)
false
(<= 1 2 3 4 5 5)
true
```

The nature of things: types

Things can be classified according to their nature, based on what they are. Forty-two is a number, `comp` is a function. In computing these classes are called *types*.

```
(number? 42)
true
(number? comp)
false
(fn? 42)
false
(fn? comp)
true
```

There are further distinctions for numbers, roughly following the types of numbers we have in mathematics.

Clojure is strongly and dynamically typed, meaning that every data item has a well-defined type which is checked when the program is running. So, you don't need to type types (no pun intended).

```
(integer? 3)
true
(rational? (/ 1 2))
true
```

However, there are some subtle differences from the mathematical classification of numbers. Numbers with decimal fractions have a different representation, bit more complicated then storing integer numbers. They are called *floating point* numbers. So 3 is the same as 3.0, but their types are different.

```
(integer? 3)
true
(float? 3)
false
(integer? 3.0)
false
(float? 3.0)
true
```

Strings

First terminology. In computing letters are called *characters*, and words and sentences are called *strings*. This is not just arbitrary naming, it's more like abstraction, by being a more general concept, we can cover more things: characters are letters, numerical digits, punctuation marks, other signs and whitespace. An elementary unit of information. In CLOJURE characters are denoted by a starting backslash.

```
(char? \a)
true
(char? \8)
true
(char? 8)
false
```

Strings are sequences of characters. The sequence can be empty, or just a few characters long, or a whole novel. They are denoted by double quotes.

```
(string? "tumbleweed")
true
(string? "")
true
(string? "Alice: How long is forever?")
true
```

Strings and characters are different things. This is easy to see as strings tend to have more than one character.

```
(= \a "a")
false
```

Similarly, a number and a sequence of characters (digits) are of different *types* of data. Therefore, they are not equal even if they in some sense represent the same quantity.

```
(= 42 "42")
false
```

We can create strings from anything by using the `str` function.

```
(str \a \b \c \space \1 2 (inc 2))
"abc 123"
```

Before we introduce other string functions, it makes sense to do a technical step to avoid typing much. For example, there is a function called `upper-case` that given a string returns another with the same letters but all capital. This function's full name is `clojure.string/upper-case`. It's rather long and it would be tedious and unreadable to type it often. After entering

What happens here is that string functions live in a different *namespace*, which is like being in a folder. We just give a more convenient access to that folder. Other programming languages might say that we load a library. Organizing your software properly into namespaces, modules, libraries is a very important part of software engineering, but it will not be discussed here.

```
(require '[clojure.string :as string])
```

we can refer to the function as `string/uppercase`, and also use any other string function happily.

```
(string/upper-case "old pond frog leaps in water's sound")
"OLD POND FROG LEAPS IN WATER'S SOUND"
(string/capitalize "i forgot.")
"I forgot."
(string/capitalize (str/lower-case "DO NOT SHOUT!"))
"Do not shout!"
```

Making memories

The difficulty of talking to a mathematician lies in the continuous demand for defining all the terms. ‘What exactly do you mean by that?’ – the oft-repeated question asks about the meaning attached to words. Same for the computer, everything has to be defined. We just call it differently: *binding*. For instance, `(+ x 1)` has no meaning unless `x` is defined.

```
(def x 2)
```

We don’t display how the REPL reports successful definitions. It varies for different REPLs.

So, `x` now refers to a value, which is its meaning.

$$\begin{tikzcd} \boxed{x} \arrow[r] & \boxed{2} \end{tikzcd}$$

Once defined, we can use the name `x`, and it will evaluate to its bound value.

```
(+ x 1)
3
```

In technical language we call the names used in Clojure *symbols*. The symbols can be just single letters, or a sequence of letters mixed with numbers and some other signs like ?,-,_, etc.. Since both symbols and strings are just sequences of characters, it is very important to distinguish them. The string `"x"` is a data literal, just a piece of data that evaluates to itself. The symbol `x` is just a name for something else, for example a piece of data or a function. It can also be undefined. Note that quoting symbols and the double quotes for strings are different.

```
(symbol? 'x)
true
(symbol? "x")
false
(string? 'x)
false
(string? "x")
true
```

Exercise 1.
How can you make `(= x "x")` to evaluate to `true`?

Though it is not the best idea in general, but meaning of symbols can be redefined.

```
(def x 10)


(+ x 1)
11
```

We can simply check the meaning bound to a name by the name itself,

```
x
10
```

but if we quote it, we get the name back.

```
'x
x
```

Just like in natural languages: apple is a delicious fruit, but "apple" is a 5-letter word. By double quotes, we can tell the someone not to find the meaning, just concentrate on the word. Quoting has the same purpose in Clojure, it is an explicit instruction for not doing evaluation. The apostrophe is just a shorthand notation. The full form of quoting looks like a function call.

```
(quote x)
x
```

However, if you have the suspicion that quoting is not really a function call (since the argument is not evaluated), then you are right. It is a *special form*.

For now it is enough to know that the purpose of special forms is to do things that are not function calls.

What is the practical benefit of binding values to names? We can store the results of partial computations, in order to save work. Instead of

```
(+ (* 2 (+ 6 7))
   (/ 3 (+ 6 7))
   (- 2 (+ 6 7)))
198/13
```

we can do

```
(def r (+ 6 7))
(+ (* 2 r)
   (/ 3 r)
   (- 2 r))
198/13
```

computing `(+ 6 7)` only once, not three times.

Where are these definitions stored? It is called the *global environment*, and it is the memory where every well-formed expressions in the REPL gets its meaning from. This can be thought of as long-term memory. This metaphor already gives a hint why is not the best idea to define everything in the global environment. We don't want to remember everything, some things we need to remember only for a short time. Later we will make definitions in local environments, in short-term memory.

List, the most fundamental collection

Programmer's life would be quite easy if there were only scalar values, like numbers. These are like the atoms of data. However, the world is more complicated than that, so the data describing everything around us comes in bigger chunks. *Data structures* are combinations of scalar data and other data structures. Simple scalar data items are the 'atoms', while compund data structures are the 'molecules' of the world of information.

Creating lists

Lists are the most fundamental data structures in Lisp-like languages. They are simple a bunch of items in a sequence inside a pair of parentheses. Looks familiar? Sure! We have been using them from the beginning. Function calls are represented as a list. Given a list, Clojure will try to call the first element as a function with the remaining elements as arguments. It is so eager to evaluate, that if we just want to have a plain list of numbers, then we need to tell Clojure explicitly to stand back and not to try to evaluate the list as a function call. We have to quote the list. Otherwise, we get an error message saying that a number is not a function.

The name Lispstands for LISt Processing.

This is the code-as-data philosophy of Lisp. The technical term is *homoiconicity*, the same-representation-ness in ancient Greek. The upshot is that programs can work on lists, and programs are written as lists, so programs can work on themselves.

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

Alternatively, we can use the `list` function that takes arbitrary number of elements and put them together in a list.

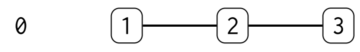
```
(list (* 6 7) (inc 100) (/ 9 3))
(42 101 3)
```

We can also construct a bigger list by combining an element and a list.

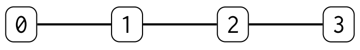
Lists prefer to connect to elements in the front. The first element is the most accessible one, for all the other we have to walk through the sequence.

```
(cons 0 (list 1 2 3))
(0 1 2 3)
```

Here we called the cons function with a number `0` and a list `(1 2 3)` freshly created by `list`,



and it returned a longer list containing `0` as well, attached to the beginning of the list.



There is no restriction on what sort of elements you can have in a list.

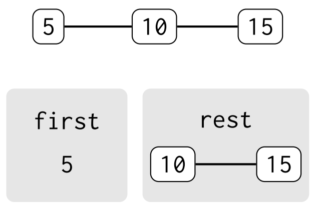
```
(cons "a string!" (list 1 2 3))
("a string!" 1 2 3)
```

The structure of lists

How about if we want to dismantle a list? That can be done along the same scheme: we separate the first element and the rest of the list.

Historically these functions `first` and `rest` were called `car` and `cdr` referring to registers in the CPU of the IBM 704 computer, in the late 1950s. And continued to be called like that long after those machines disappeared. Sometimes it is nice to break with the tradition.

```
(first '(5 10 15))
5
(rest '(5 10 15))
(10 15)
```



The name of nothing: `nil`

Now, a tricky question. What is the first element of the empty list? Well, it's nothing. But remember, the computer needs precision, we have to be exact even when we talk about nothing. Therefore nothing, the void, the vacuum, the nonexistent, the empty, the oblivion is called `nil`.

```
(first ())
nil
```

Note that the empty list needs no quoting, since there is no danger to take the first element as a function. Only nothing is nothing, everything else is something.

```
(nil? 0)
false
(nil? ())
false
(nil? false)
false
(nil? nil)
true
```

What is the rest of the empty list? We agreed that `rest` will return a list no matter what. Also, we cannot produce a non-empty list from thin air. That leaves only one choice.

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

Size of a list

We can ask for the number of elements contained in a list.

```
(count '("a" \b 1.2))
3
```

There is a predicate for deciding whether a list is empty or not.

```
(empty? ())
true
(empty? '(1 2))
false
```

Exercise 2.
If `empty` was not available, how can we test for emptiness?

Concatenation

We can build lists by using other lists as building blocks, by connecting them in a given order.

```
(concat '(1 2) '(3 4))
(1 2 3 4)
(concat '(3 4) '(1 2) '(5 6))
(3 4 1 2 5 6)
```

`concat` can take arbitrary many arguments, including one and zero.

```
(concat '(\a \b c))
(\a \b \c)
(concat)
()
```

Concatenating a single list is just that list. Concatenating nothing is just another way to produce the empty list.

`reverse` *and* `last`

Here are two self-explanatory functions for lists.

```
(reverse '(1 2 3 4))
(4 3 2 1)
(last '(5 6 7))
7
```

But what if only `reverse` was available. Could we somehow get the last element of the list without `last`? The last element is the same as the first of the reversed list. We have `first`, so simple function composition works.

This is a recurring theme in the learning process. Pretend that some existing function is not available, then write it.

```
((comp first reverse) '(5 6 7))
7
```

Exercise 3.
What does the composite function `(comp last list)` do?

Lazy sequences of numbers

We often need to produce an ascending list of numbers, therefore this task is automated. The `range` function can produce numbers starting from zero up to a limit.

```
(range 13)
(0 1 2 3 4 5 6 7 8 9 10 11 12)
```

The limit is not included in the result, so `(range 0)` is a synonym for the empty list.

When two arguments are given to `range`, they are interpreted as the start of the sequence and the end of the sequence.

```
(range 1 10)
(1 2 3 4 5 6 7 8 9)
```

The third argument can change the step, which defaults to one. We can get produce the even and odd single digit numbers.

```
(range 0 10 2)
(0 2 4 6 8)
(range 1 10 2)
(1 3 5 7 9)
```

`range` is not limited to integer numbers.

```
(range 1.1 4.2 0.5)
(1.1 1.6 2.1 2.6 3.1 3.6 4.1)
```

Up to this point `range` gave nothing surprising. But what happens when we give no argument at all? The start point defaults to zero, but what is the limit? Well, there is no limit, or as mathematicians would say, infinity is the limit. Without arguments, `range` returns the list of all natural numbers. The following definition is valid,

```
(def natural-numbers (range))
```

and the binding is done immediately. The computer now has $\mathbb{N}$, the infinite set of natural numbers bound to the symbol `natural-numbers`.

How is this possible? How can we fit infinitely many numbers into the machine that has finite memory? Being *lazy* is the solution. Not doing anything until the last minute, unless explicitly asked. If we force the REPL to print `natural-numbers`, it will sooner or later produce some error message complaining about memory not being sufficient enough.

It is a good idea to try this. One has to know how to stop a computer program when it goes rogue.

What is the purpose then? It is useful not to set artificial limits to computations. We can do computations on demand, so we can deal with arbitrary big instances of a problem, memory permitting of course, but without doing any administration to change limits. Laziness is a crucial idea, its usefulness will become apparent later. For now, let's see how we can deal with an 'infinite list'. We can take the first five elements.

```
(take 5 natural-numbers)
(0 1 2 3 4)
```

We can also take the second five elements by dropping the first five first (still producing a lazy infinite list).

```
(take 5 (drop 5 natural-numbers))
(5 6 7 8 9)
```

When we don't know how many elements are needed, we can make it the taking and dropping more flexible by giving a predicate.

```
(take-while neg? [-2 -1 0 1 2])
(-2 -1)
(drop-while neg? [-2 -1 0 1 2])
(0 1 2)
```

`take-while` collects the elements from the original sequence as long as the predicate returns `true` when applied to the elements and stops when it gets `false`; `drop-while` discards elements as long as the condition is satisfied, then returns the rest.

Vector, sequential collection that is also a function

Vectors are sequences of some elements enclosed within square brackets. On the surface they very much look like lists, but with a different delimiter. We'll see later that differences between lists and vectors are fundamental. We can create a vector the short way as a data literal

Using other delimiters beyond parentheses is a great innovation of Clojure.

```
[1 2 "three"]
```

or the long way, by using the `vector` function

```
(vector 1 2 "three")
[1 2 "three"]
```

or we can turn another collection (here a list) into a vector.

```
(vec '(1 2 "three"))
[1 2 "three"]
```

Even just on the surface, vectors are great because we don't need to quote them. A vector is not a list so no one tries to evaluate it as a function call. So whenever we want to write down a sequence of elements, we'd better use vectors. The differences between lists and vectors are lot deeper than this. In a list we can access its elements one-by-one, walking through its structure by `first` and `rest`. If we want to get the last, we have to visit each element. Vectors are more accessible. They are *indexed*, meaning that each element has an associated integer number starting with `0`. For instance, the vector `["a" "b" "c"]` can be visualized as a lookup table.

0	1	2
"a"	"b"	"c"

It is like a function that produces elements for index numbers. Indeed!

In mathematics we say the vector is a map from $\mathbb{N}$, the set of natural numbers to a set of objects (the things we can store in a vector).

```
(["a" "b" "c"] 0)
"a"
(["a" "b" "c"] 2)
"c"
```

Now this is something that really widens the concept of a function! A data structure that doubles as a function. Yet another example of data-as-code. A weird puzzle:

```
((comp ["a" "b" "c"] [2 1 0]) 0)
"c"
```

Why?

There is of course a more pedestrian way to get the values from a vector. We can just ask for the *n*th element.

```
(nth ["foo" "bar"] 0)
"foo"
(nth ["foo" "bar"] 1)
"bar"
```

Accessing the elements of vectors is not as forgiving as `first`, `rest` for lists. Using an index goes beyond the elements in the lists results in an error.

Constructing a bigger vector can be done by conjoining an element.

```
(conj [11 13 17] 19)
[11 13 17 19]
```

Vectors like to connect at the end and it is easy to see why. Connecting anywhere else would mess up previous associations. Note that the order of the arguments for `conj` reflects this, just as in `cons` for lists.

Immutability

Consider the following little conversation with the computer.

```
(def v [1 2 3])
v
(conj v 4)
[1 2 3 4]
```

Now the question is what is the current value of `v`? What does `v` evaluate to? If your answer is `[1 2 3]` (the correct one), then there is nothing to worry about. We associated the vector `[1 2 3]` with the symbol `v`, then we called the `conj` function with arguments `v` and `4`, which returned the new combined vector `[1 2 3 4]`, but the association of `v` has not changed.

However, if you say `v` is `[1 2 3 4]`, then you must have some previous programming experience in a programming language with mutable data structures, and CLOJURE's *immutable* data structures can look a bit strange in the beginning. Both approaches have advantages and disadvantages, however if we run programs concurrently or in parallel then immutable data structures have a clear edge.

So what is the immutability exactly? Simple data items never change. 42 is 42 forever. I can add one to it, `(inc 42)` equals 43. Function create new values but do not touch their arguments. Naturally, 42 will not mutate into 43 just by applying the function $f(x) = x + 1$ to it. This is so obvious, that it feels a bit silly to mention it.

However, for composite data items immutability is less obvious. It makes perfect sense to add a new element to a vector, by changing it to a bigger one. However, in CLOJURE a new vector is created and the old one is retained. Well, it's better than that. It is also efficient: there is structural sharing between the old and the new vector. There is no unnecessary duplication of the data items stored in vectors.

This is exactly how version control systems work, e.g. GIT. They store only the difference between different versions of the documents.

Processing sequences in one go: `map`, `apply`

The upshot of having lists is that we can pack some elements together and handle them as a single thing. Really. If you want to call the same function for all elements of a list, you don't need to bother with doing it one by one. We just 'map' a function across the elements of a collection:

`(map f [x y z])`

will evaluate to

`((f x) (f y) (f z))`

The archetypal example appears in every functional programming introduction.

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

Well, the name may not be the best, but the idea should be simple enough to be clear. Is `map` restricted to single-argument functions? No.

```
(map + [1 2 3] [40 50 60] [700 800 900])
(741 852 963)
```

It takes the first element from each supplied collection to make the arguments of the first call of the mapped function, then the second elements, and so on.

How about the opposite case? What to do when we have elements in the list, and we want to call a function with those as arguments, but not with a single list. For instance, addition expects a bunch of numbers and it would not work with a list. The solution is `apply` that can feed the elements of a list properly to a given function:

`(apply f [x y z])`

evaluates to

`(f x y z)`

So, `(+ [10 11 12])` gives an error message, since the `+` function expects numbers and doesn't know what to do with a list.

```
(apply + [10 11 12])
33
```

There is a clear distinction between a function taking *n* arguments and another taking a list with *n* elements. For the latter there is only one argument.

The function `str` converts its arguments into strings and concatenates them into one big string.

```
(str [\h \e \l \l \o])
"[\\h \\e \\l \\l \\o]"
```

The answer is correct, `str` turns the vector into a string, but it may not be what we expect. If we want to turn a sequence of characters into a string, then we need to use `apply`.

```
(apply str [\h \e \l \l \o])
"hello"
```

`apply` accepts multiple arguments, but only the last argument will be treated as a collection containing more arguments.

```
(apply str "Hello" " " [1 2 3] " " [\w \o \r \l \d \!])
"Hello [1 2 3] world!"
```

Selecting things from sequences

Selecting things from a sequential collection is nicely automated. Imagine we have a vector full of all kinds of stuff, and we need to separate the elements by their nature, by their type. The function `filter` takes a predicate (a yes-or-no question) and a sequential collection as its second argument. Then it goes through the collection, applies the predicate to all of its elements, selects those that give a yes answer, and finally returns a newly built list of selected items. We can separate the elements by separating by their types.

People coming from programming languages of the imperative style may ask 'Where is my `for`-loop?'. The short answer is that there is no need for them. Higher order functions like `map` and `filter` (and other constructs later) give the same functionality of going through a collection and doing something to the elements; with the added bonus that we don't have to fiddle with a loop variable.

```
(def v [-1 0 2 "two" 'foo 42 "42" -6])

(filter number? v)
(-1 0 2 42 -6)
(filter string? v)
("two" "42")
(filter symbol? v)
(foo)
```

Since filter also returns a collection, we can do filtering again on that to get some finer selection.

```
(filter even? (filter number? v))
(0 2 42 -6)
```

People seeing `map` and `filter` for the first time might have some problems distinguishing between the two. So it is instructive to put them side-by-side.

```
(map pos? [-2 0 1])
(false false true)
(filter pos? [-2 0 1])
(1)
```

If we map a predicate over a collection, we get a list of logical values. When filtering, based on the returned value of the predicate applied to an element, we decide whether we need to put the element into the result collection or not. The size of the result for `map` is the same as the input collection, for `filter` it can be the same size (predicate says true for all), smaller, or even zero, when there is no positive answer.

Sometimes we want to select the elements for which the predicate returns `false`.

```
(remove zero? (filter number? v))
(-1 2 42 -6)
```

The name of the function is a bit misleading, by immutability, no elements are removed from vector `v`. A new list is created for the numbers, leaving out zero.

Hash-maps

Association is a mental connection between things and ideas. The most common example is naming. We associate meaning to words, that are sequences of letters.

In the wake of introducing vectors, we should do a powerful abstraction. A vector associates things, some *values* to numbers $0, 1, 2, 3, \ldots$, to the so-called *keys*. This is useful for some things in an ordered sequence, but it is a very special association. What if the keys can be anything? Here are some wild associations.

```
{"answer" 42 "question" nil 2 "two" "list" '(1 2) () '(3) }
{"answer" 42, "question" nil, 2 "two", "list" (1 2), () (3)}
```

Curly braces indicate a *hash-map*. The name is unfortunate, since it describes how these associations are implemented, and not what they do. Lookup table might be a better name, but we are stuck with the old name. The computer politely puts commas after the each key-value pair. We can do that as well, but it is not necessary. We associate the string `"answer"` with the number `42`, while the question itself is unknown, so the key `"question"` gets associated with `nil`. (Yes, these associations are from an old science-fiction book.) Then number `2` is connected to its English name, the word `"list"` is associated with a two-element list of numbers, then the empty list with a one-element list.

So anything can be used as keys and values. Even nothing is permissible as a key: `{nil "nothing"}` is a permissible map. But there is a type of keys, that is specially designed for hash-maps: *keywords*. They are symbols starting with a colon. So, what was the book you mentioned?

Here the input spreads over several lines. This is no problem, CLOJUREwill no that you are not finished when you hit ENTER, since opening parens are not closed yet. When the definition is finished, it will acknowledge

```
(def book {:author "Douglas Adams"
           :title "Hitch-hikers' guide to the galaxy"
           :year 1979})
```

This is an all-time favorite science-fiction novel with computers playing central roles. [2]

A hash-map behaves like a small database. You can look up values by giving keywords.

```
(book :author)
"Douglas Adams"
```

A database as a function?!? Yes, hash-maps can be used as functions with a key as the single argument. The result of the function call is the corresponding value, or `nil` if it is not in the map. There are more neat surprises.

```
(:author book)
"Douglas Adams"
```

Keywords also behave as functions! They look themselves up in a map.

Exercise 4.
What is the output?

Just a reading exercise! This is by no means a good coding style!

```
((comp {:b 11 :a 13 :c 17} [:c :b :a] [1 3 2]) 2)
```

There is an elegant way to create a hash-map when we have the keys and values in different sequential collections, but matched up nicely. We can just 'zip' them together.

```
(zipmap (range 4) ["zero" "one" "two" "three"])
{0 "zero", 1 "one", 2 "two", 3 "three"}
```

There is also easy access to the keys and values separately.

```
(keys {:x 1 :y 11 :z 111})
(:x :y :z)
(vals {:x 1 :y 11 :z 111})
(1 11 111)
```

Sets

In a hash-map, keys cannot be duplicated (otherwise the output value of the lookup function would be ambiguous). Imagine a hash-map where all keys are mapped to themselves. Then basically we have a *set* in the mathematical sense.

```
(def numbers #{1 2 3})
```

Hash-sets can be defined by curly braces prefixed by a #. As in mathematical sets, the order of the elements does not matter.

```
numbers
#{1 3 2}
(numbers 4)
nil
(numbers 3)
3
```

Looking up the elements works similarly, hash-sets are also functions.

```
(def fruits #{:apple :mango :pear :grape :banana :orange})
(def vegetables #{:cucumber :tomato :carrot :onion})
(def basket [:apple :apple :onion :carrot :grape])
(filter vegetables basket)
(:onion :carrot)
(filter fruits basket)
(:apple :apple :grape)
```

The sequence abstraction

All the collections so far, namely lists, vectors, strings, hash-maps, hash-sets can be handled as sequences of elements. We can use the seq function to investigate how the collections are turned into logical sequences. Being a sequence is not a surprising property for lists and vectors, since they are sequential data structures anyway.

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

Strings are also sequences, whose elements are characters.

```
(seq ”hello”)
(\h \e \l \l \o)
```

For hash-sets, the elements can be enumerated one-by-one, but it is important to note that there is no promise about the order of elements.

```
(seq #{:a :b :c})
(:c :b :a)
(seq #{1 2 3 5 8 13})
(1 13 3 2 5 8)
```

How about hash-maps? They don’t really have single elements in them, rather contian pairs of elements. These key-value pairs van be represented by vectors. Accordingly, the sequence of a hash-map consists of key-value pairs in vectors.

```
(seq {:x 1, :s ”foo”, :c \x})
([:x 1] [:s ”foo”] [:c \x])
```

So it seems that the right way to think about hash-maps that it is a collection that contains associations, treated as single things.

```
(count {:a 11, :b 13})
2
```

This also shows that there is no need to call `seq` explicitly. Clojure takes care of turning the collections into sequences, whenever needed. This way, all the sequence functions like `first`, `rest`, `first`, `second`, `last` and the higher order sequence processing function like `map`, `filter`, etc. can work with all the above data structures.

What is the point? The sequence abstraction can be justified in different ways. From the students' perspective it is a big win, since one only needs to remember functions dealing with sequences, then working with other data structures can be done without learning anything new. The software engineer relishes the fact the same piece of code would work in many different situations, enabling *code reuse*.

Here the unmissable quote is: "It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.", by Alan J. Perlis in the foreword of SICP [1].

Where is the catch? The behaviour of the same function can be different for different data structures. For example, the difference between `cons` and `conj` can be confusing. `cons` builds a sequence, always putting a new element in the front,

```
(cons 1 '(2 3))
(1 2 3)
(cons 1 [2 3])
(1 2 3)
```

while `conj` *conjoins* a new element in a way that is 'natural' for the collection, (front for lists, back for vectors)

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

and it can take arbitrary many arguments.

```
(conj [2 3] 4 5 6)
[2 3 4 5 6]
```

Conjoining keeps the type of the collection.

```
(conj {:name "Arthur"} [:age 42])
{:name "Arthur", :age 42}
```

Beyond the different behaviours, performance can also be an issue. One can use `nth` on lists, but it will not be as efficient as on vectors, since Clojure has to hop through all the preceding elements.

Defining functions

In a more traditional introduction to programming, defining functions would be the second step, right after the arithmetic operators. In CLOJURE we have many ways of combining functions and the associative data structures also behave as functions, thus the issue is less urgent. But this does not diminish the significance of crafting functions.

One way to think about functions is that they are great time-saving tools. Same as in algebra, we use a single letter to denote a multitude of choices of things, there most notably numbers. For example, when talking about square numbers, I can mention a few: $5 \cdot 5 = 5^2 = 25$, $11 \cdot 11 = 11^2 = 121$. I can also refer to all square numbers by writing $x \cdot x = x^2$ when x is any natural number. Same happens in programming. We can do concrete calculations,

```
(* 2 2)
4
(* 12 12)
144
```

but soon we would get tired of typing the numbers twice. Alternatively, we can forget about the concrete numerical values, replacing them by a symbol, for instance `x`. This `x` is a hole to be filled later. Thus, we get an *abstract expression* `(* x x)`, but in itself this does not make sense, since what is `x`? We need to attach some meaning to it. For instance, after `(def x 2)` the expression `(* x x)` evaluates to `4`; after `(def x 12)` to `144`. So far so good, but there are two major problems with this approach. First, we have to be very disciplined by using the symbols: the symbol in the definition has to match the symbol in the abstract expression. Second, we make permanent changes to the environment by attaching meanings to symbols. This may seem harmless first, but in practice the effects could render interacting with the computer totally useless. Imagine two abstract expressions using the same symbol, but expecting different meanings – recipe for disaster.

The official term for a 'hole' in an abstract expression is *free variable*.

Humans are really bad at administration. If we have to write the same thing at two different places, sooner or later we will write something else and get genuinely surprised by inconsistent results.

There is of course a better mechanism for making the abstract expression reusable, and that is the same as creating functions. We plug the whole by saying that it is the input to the function. We need to say that a value for x will be supplied later in the abstract expression by writing (fn [x] (* x x)). This reads as a function that takes a single input value and returns its square.

```
((fn [x] (* x x)) 2)
4
((fn [x] (* x x)) 12)
144
(map (fn [x] (* x x)) [1 2 3 4 5])
(1 4 9 16 25)
```

In all these examples we create a function, apply it to some argument(s) and throw it away. Coming up with good names for functions is often mentioned as the single most difficult problem in software engineering. So, it is nice to have the option of not naming them when they are short and self-explaining.

The official term is *lambda function* for anonymous, throw-away functions. See the Clojure logo.

Of course, we can make memories, where the value attached to a symbol is a function. We can make a definition (def square (fn [x] (* x x))), but this comes up so often so there is a shortened way to define named functions.

```
(defn square
  [x]
  (* x x))
```

The special form defn reads as 'define function'. The name of the newly defined function is square. The following vector contains the input parameter of the function (here just a single x); then finally comes the *body* of the function: an abstract arithmetic expression. After this function definition, we can calculate square numbers in an elegant way.

```
(square 2)
4
(square 12)
144
```

When the square function is called with number 2, the function's input parameter x is bound to 2, so evaluating its body gives 4. One can think of as substituting a concrete value into x. Important to note, that this assignment is temporary, just valid for the function call. So even if x is defined in the environment, it has no effect on the function's value.

```
(def x 3)

(square 10)
100
x
3
```

There is a *global* `x` and a *local* in the function. The function has its own little environment.

The real great thing about functions, that after writing them we can forget about their details. They are indistinguishable from Clojure's own functions. Therefore, we can use this newly defined function as many times I want, no restrictions.

```
(map square [2 5 11 100])
(4 25 121 10000)
```

It is concievable that a function does not have an input argument.

```
(defn greetings
  []
  "Hello world!")
```

There is no input to depend on to have different output values, so this is a *constant* function. `(greetings)` will always evaluate to `"Hello world!"`. Also, the function body is quite simple, just a string literal.

There can be more arguments.

```
(defn rectangle-circumference
  [a b]
  (* 2 (+ a b)))

(rectangle-circumference 3 2)
10
```

Lambda function shorthand notation

Anonymous functions save us from the burden of naming them, but we still need to do some naming. We have to give names to the inputs of the function. Luckily this can be avoided as well.

```
(map #(* % %) [1 2 3 4 5])
(1 4 9 16 25)
```

The hashmark indicates a function literal and the `%` symbol refers to the argument of the function. If there are more arguments then numbering can be used `%` or `%1` for the first argument, `%2` for the second, and so on.

Exercise 5.
Define the function $f(x) = 4x^2 + 2x - 4$ in Clojure!

Decision making – conditionals

We need to be able to make decisions, explicitly. Without the ability of choosing between alternatives, we only have a fancy symbolic calculator. The simplest way of making a choice is the if-then-else form.

Having conditionals is a decisive feature, as it was discovered by Charles Babbage in 1837. He switched from the Difference Engine project to the Analytical Engine, but never finished it. This probably delayed the birth of computers by a century.

```
(if true "consequent" "alternative")
"consequent"
(if false "consequent" "alternative")
"alternative"
```

This is a special form, not a function call, since it has a different evaluation strategy. Not all elements in the form are evaluated. The condition after `if` is evaluated first. If true, then the consequent is evaluated and that is the value of the `if` form (and the alternative not evaluated at all - why bother if its value is not needed?). If the condition is false the alternative is evaluated, giving the value of the form.

In the above examples the conditions and the choices are data literals. In a less artificial situation we can replace them with symbols and function calls. Given that the symbol `a` is bound to some numerical value, we can produce its absolute value by

```
(if (< a 0) (- a) a)
```

Exercise 6.
Is there a way to calculate the absolute value without `if`?

Let's say we would like to calculate $a + |b|$, where $a, b \in \mathbb{R}$. With `a` and `b` bound to numerical values and using the definition of absolute value, the straightforward, 'prosaic' way to write the expression is

```
(if (> b 0)
    (+ a b)
    (- a b))
```

as we need to choose between two cases. There is a 'poetic' way as well, in a sense that we can express a deep truth with very few words.

```
((if (> b 0) + -) a b)
```

Here we emphasize that functions are first class citizens, they are treated as any other values.

The logic of truthy and falsey

How to express more complicated conditions? Just as in everyday language, we can negate statements, connect them with and and or. However, what is considered to be true or false has a generalized meaning in Clojure. **Both `false` and `nil` are treated as false, every other value is treated as true.**

```
(if true "consequent" "alternative")
"consequent"
(if 0 "consequent" "alternative")
"consequent"
(if () "consequent" "alternative")
"consequent"
(if [] "consequent" "alternative")
"consequent"
(if + "consequent" "alternative")
"consequent"
(if nil "consequent" "alternative")
"alternative"
(if false "consequent" "alternative")
"alternative"
```

So logical expressions can be *truthy* or *falsey*. Note this does not turn `nil` into `false`, or `1` into `true`,

```
(true? 1)
false
(true? true)
true
(false? nil)
false
(false? false)
true
```

but in decision making situations we are quite liberal about what counts as truth. What are the benefits? The flexibility in expressing selections,

```
(filter #{1 2} [:a 1 :b 2 2 :a :a 1 1 2])
(1 2 2 1 1 2)
```

using that a set used as a function returns `nil` if the argument is not in it.

Truthiness is also good for the efficient evaluation of logical expressions. The special form `and` evaluates its arguments until it find a falsey one,

Short-circuit evaluation is the computer science term for trying to save work when making decisions.

```
user=> (and (< 4 5) false (range))
false
user=> (and (= 1 (inc 0)) nil (range))
nil
```

note that the dangerous `(range)` is not evaluated at all; or if they are all truthy it returns the last.

```
(and (< 4 5) [] [1 2])
[1 2]
```

The returned value may feel strange, but an `if` statement would interpret this as a yes. The form `or` does the opposite, it returns the first truthy, or the last falsey if they are all falsey.

```
(or nil false 1)
1
user=> (or false  nil)
nil
```

Negation also accepts truthy and falsey values, and produces geniune `true`/`false` answers.

```
(not nil)
true
(not "hello")
false
```

Iteration

We often need to call a function *iteratively*, i.e. feeding the output back into the function. Let's consider a simple mathematical game. Given a positive integer, if it is even then half it (still a whole number), when it's odd multipply by 3 and add 1. What happens if you do the same with the result? Again and again. The function itself is easy to write.

```
(defn c [n]
  (if (even? n)
    (/ n 2)
    (inc (* 3 n))))
```

So we can try.

```
(c 1)
4
(c 4)
2
(c 2)
1
```

From 1 we get back to 1. From 8 we would go to 4, then again get back to 1. Is this the case for other numbers? Interesting problem but tedious to call the function again and again. So, let's automate! In algebraic notation, we want the sequence

In mathematics this is called the Collatz conjecture, and it is a particularly tough problem. We do not know at the time being whether for all numbers the process eventually returns to 1 or not.

$$n, c(n), c(c(n)), c(c(c(n))), \ldots$$

we want to repeat the function c, calling it with the newly produced value. The function `iterate` does that exactly:

`(iterate f x)`

produces the lazy list

`( (f x) (f (f x)) (f (f (f x))) ... )`, so in particular we can use the above function `c`

```
(iterate c 1)
```

But when you hit enter, no result. Seemingly. The computer evaluates `c` with argument `1`, then takes the result as an argument of `c`, infinitely many times (at least until the computer's memory fills up). That's why it never returns a value. This is another example of dealing with something potentially infinite. The functions `take` and `drop` can tame infinite sequences of numbers, and they can do that to infinite sequence of function calls. We simpy ask to return the first few values of the infinite sequence.

```
(take 20 (iterate c 9))
(9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1)
```

We can also ask what is the sixth element of the sequence (indexed by 5)?

```
(first (drop 5 (iterate c 9)))
11
```

Exercise 7.

The `range` function called without arguments produces the infinite lazy list of natural numbers. Can we produce the same list without `range`?

Recursion

The simplest way of defining *recursion* is that a function calls itself. It may even look paradoxical, since we are defining something in terms of itself. The paradox disappears when we realize that when function calls itself it does it with arguments that somehow represent a smaller and easier to solve version of the problem. So recursion is a way of dividing a difficult problem into easier ones. For instance, when a problem is somehow given as a list we can naturally divide it into two parts: dealing with the first element, and dealing with the rest of the list. Recursive functions are list are the classical examples of functional programming.

Self-reference is where things get interesting. Consider the sentence "This sentence is false."; it does not have a well-defined logical value. Beyond logical paradoxes, self-reference is often thought to be a key ingredient to consciousness. See for example [3].

Let's imagine `count`, a function for getting the size of a collection, is not available, so here is an implementation for lists.

Pretty, but not the most efficient way, in practice we will use reducing (folding).

```
(defn COUNT
  [l]
  (if (empty? l)
      0
      (inc (COUNT (rest l)))))
```

If the list is empty, then the function returns `0`. This is the base case, where the recursion stops. Otherwise, it knocks off the first element of the list and calls itself for the rest of the list. This way it calculates the size of the list by adding one to the size of the list without the first element.

We use uppercase name for the new implementation to avoid clashing with the existing `count` function.

Another classic is the re-implementation of `map`, again dividing the work by `first` and `rest`.

```
(defn MAP
  [f coll]
  (if (empty? coll)
    ()
    (cons (f (first coll))
          (MAP f (rest coll)))))
```

Making short-term memories

How about short-term memory? What if we want to use some binding, but only temporarily? Just until we evaluate an expression. `let` lets us do that.

```
(let [r (+ 6 7)]
  (+ (* 2 r)
     (/ 3 r)
     (- 2 r)))
198/13
```

The temporary bindings are put into a vector. There can be more bindings, but they should come in pairs.

The bindings are done in order, so we can use a binding immediately to define another one.

```
(let [x 1
      y (inc x)
      z (+ x y)]
   [x y z])
[1 2 3]
```

Typically we use `let` bindings in functions, to get a better structuring of the computation to be done.

Destructuring

How often do we get some collection as an argument of a function and then spend half of the body of the function to pull the collection apart? Destructuring handles this. Just like kicking the ball on the volley, we can give names to parts of the arguments before they land in the function.

For example, we a represent a line defined by two points by a vector of vectors and we would like to compute the slope of the line like `(slope [[1 2] [2 4]])`. So we define the function `slope`.

```
(defn slope
  [line]
  (/
    (- (first (first line)) (first (second line)))
    (- (second (first line)) (second (second line)))))
```

Here the coordinate information is extracted on demand, making the actual calculation obscure, littered with the retrieval. We can separate these two.

```
(defn slope2
  [line]
  (let [p1 (first line)
        p2 (second line)
        x1 (first p1)
        y1 (second p1)
        x2 (first p2)
        y2 (second p2)]
    (/ (- x1 x2) (- y1 y2))))
```

Now the computation is quite clear, it is basically the mathematical formula. However, we have a long list of bindings. Destructuring gets rid of this, by giving the ‘shape’ of the input data in the argument list.

```
(defn slope3
  [[[x1 y1] [x2 y2]]]
(/ (- x1 x2) (- y1 y2)))
```

This may look like magic first, but it is actually just a simple automation. It is easy to reveal how it is done.

`destructure` is very useful in figuring out what goes wrong in an unsuccessful and complex destructuring attempt.

```
(destructure '[ [x y] [13 19]])
[vec__1246 [13 19]
 x (clojure.core/nth vec__1246 0 nil)
 y (clojure.core/nth vec__1246 1 nil)]
```

It does exactly the work we did not want to do manually. `destructure` produces a vector of bindings, that is given to `let` in a real destructuring situation.

Reducing a collection

Often we need to make a single value out of a collection. Or another collection, for that matter. Reducing is a very general operation, therefore it may be perplexing first. A visual image may help to get the idea. Imagine a child collecting pebbles on the seashore. Picking up stones with the right hand, while holding the already collected ones in the left palm, or in a small bucket hold in the left hand. Or the child can decide not to gather all pebbles, just the most beautiful one. For each newly picked up pebble, she compares it with the one held in the left hand, and decides which one to keep. In both cases, some result is accumulated in the left hand.

Let's see how to collect pebbles in Clojure, or rather keywords into vectors.

```
(reduce conj [:a :b]  [:c :d :e])
[:a :b :c :d :e]
```

Here we have an initial collection `[:a :b]` (the pebbles already in the bucket), and the keywords `[:c :d :e]` to be collected (the pebbles still on the beach). 'Putting a pebble into the bucket' is done by `conj`ing them one by one to the existing collection. The fact that we only see the result may hinder understanding the reduction process. Luckily, it is possible to see the accumulation step-wise.

```
(reductions conj [:a :b]  [:c :d :e])
([:a :b] [:a :b :c] [:a :b :c :d] [:a :b :c :d :e])
```

Reducing some elements into a new collection is such a common operator that there is a dedicated function for that. Roughly speaking `into` is just `reduce conj`.

```
(into [:a :b]  [:c :d :e])
[:a :b :c :d :e]
```

Now for finding the most beautiful pebble. This is the same problem as finding a maximal number from a collection of numbers.

```
(reduce max [1 2 1 3 2 5 4 6 1 2])
6
(reductions max [1 2 1 3 2 5 4 6 1 2])
(1 2 2 3 3 5 5 6 6 6)
```

Here we didn't specify the initial value of the reduction, the first value of the vector can serve as the initial value.

At this point it might be difficult to see the difference between `apply` and `reduce`, since in special cases they produce the same result.

```
(reduce + [1 2 3 4 5 6 7 8 9])
45
(apply + [1 2 3 4 5 6 7 8 9])
45
```

However, the shape of the process is different: `apply` yields `(+ 1 2 3 4 5 6 7 8 9)` while `reduce` will go through steps `(+ 1 2) (+ 3 3) (+ 6 4) (+ 10 5) (+ 15 6) (+ 21 7) (+ 28 8) (+ 36 9)`. What happens is that reduction requires a function with 2 arguments, the so called *reducing function*. Its first argument is an *accumulator*, some data in which we collect the result of the computation. The second is an element from the collection being processed by `reduce`. Here is a simple example of a reducing lambda function: `(fn [c _] (inc c))`. This ignores its second argument, but whenever it gets a new one, it increments the counter. This can be used for re-implementing `count`.

The + function with multiple arguments will do reductions internally anyway, so ultimately they realize the same process.

```
(defn COUNT
  [coll]
  (reduce (fn [c _] (inc c))
          0
          coll))
```

Note that this implementation is not recursive.

The general form of `reduce` is this.

```
(reduce (fn [accumulator thing]
          (process accumulator thing))
        initial-value
        collection)
```

The initial value is optional, in case it can be inferred from the collection.

Point-free style

We can create new functions by specifying what they do with their arguments, but we can also make new functions without mentioning those input values (points). There are several ways to do this:

- composing functions by `comp`,
- preloading arguments by `partial`,
- grouping functions to work on same input by `juxt`,
- negating logical output value by `complement`.

Nested function calls of the form `(f (g (h x)))` can be replaced by `((comp f g h) x)`.

```
(def negative-product (comp - *))
(negative-product 2 3)
-6
```

With `partial` we can create functions by preloading the first few arguments of a function with several inputs. In a sense we turn a general function into a more specific one.

```
(def ten-times (partial * 10))
(map ten-times [1 2 3])
(10 20 30)
```

When we want to apply several functions to same input, we can `juxt`apose them. `(juxt f g h)` is the function that produces `[(f x) (g x) (h x)]` when applied to `x`. It works functions that can take more arguments, but obviously they need to have the same number of inputs,

```
((juxt take drop) 3 [1 2 3 4 5 6])
[(1 2 3) (4 5 6)]
```

and `juxt` always returns the result in a vector.

One thing where `juxt` really shines is retrieving data items from a hash-map, more than one at the same time.

```
(def book {:author "Terry Pratchett"
           :title  "Hogfather"
           :year   1996
           :series "Discworld"})
((juxt :title :year) book)
["Hogfather" 1996]
```

Combining these higher-order functions can be very expressive. Here is a one-liner function for segregating even and odd numbers.

```
(def sgr (juxt (partial filter even?) (partial filter odd?)))
(sgr (range 9))
[(0 2 4 6 8) (1 3 5 7)]
```

It's amazing how far one can get with point-free style. Here we define a function that counts the number of lowercase letters in a string.

Of course, the standard solution for this problem would use regular expressions.

```
(def count-if (comp count filter))
(def letters (set "abcdefghijklmnopqrstuvwxyz"))
(def count-lowercase (partial count-if letters))
(count-lowercase "aBbC10x")
3
```

It is debated how desirable is the point-free style in practice. Some say it is 'pointless', and readability can be a subjective issue.

How to code it?

Trying to solve a programming problem can be a daunting experience. Not knowing how to start, or not understanding why the code is 'misbehaving', no clue about the next step, these are intimidating and frustrating. On the other hand, programming is not magic, anyone with a bit of patience can learn it. Here are a few pieces of advice, that may (or may not) help overcoming these difficulties.

Programming and solving mathematical problems have a lot in common, so the classic book 'How to Solve It' [5] should be recommended here.

It's OK to be confused.

Really. Abstract thinking is a defining characteristic of human beings, but in programming we take this ability to extremes, so it does not come that easily.

Professional programmers also have the feeling of being lost frequently. For example, even if someone is a master of a language, still, there is a need to work with someone else's code, and exploring the unknown is not a straight process. Being professional means that you know how to recover quickly by going back to the point where you got lost.

This applies to the whole Clojure learning process as well. If you loose track of what's going on. Just go back to the basics: programs are written list denoted by parens, the first element is the operation. . .

Practice

It is not a big secret. If we keep doing something, we get better at it. Solving more and more problems makes you a better programmer.

What is less known is that solving a problem only once is often not enough. Performing musicians do not stop practising after they could play the whole piece without mistakes for the first time. They play it again, to reinforce the memory and to improve the playing style. Same for programming. You solve a problem, for the first time, probably with the help of someone or the Internet. Then you should clear the screen and solve the problem again – this time alone. And again. Repeat this a few times, then you will see a better way to do it. Eventually, you get bored, and that is a good sign. You can move to a harder problem.

Play in the REPL!

This is a repeated advice, but what does it mean exactly? Playing with what?

Playing in a sense of just trying things out, just reminding ourselves how data structures behave under certain functions.

What are the data structures mentioned in the problem? Characters and strings? Then let's just turn a bunch of characters into strings, or blow up a string into a sequence of characters.

Learn to throw away code

For small problems, it is a good idea to restart problem solving a couple of times. We are bound to make mistakes, and it is often difficult to understand what goes wrong. Debugging takes a long time. After erasing everything, we might have a good chance of not making the same error again.

If you keep code that is not really working, then it becomes part of the problem to be solved. You may need to make a lot of effort to code around a previous mistake, trying to neutralize it. You basically replace the original problem with a lot more difficult one.

In software engineering, the practise of throwing code away is very much needed too. Unfortunately, it is often impossible. This happens for instance when come not so good code is in use already. This the problem of *legacy code*.

Wishful thinking

Problems seldom can be solved in one go. Don't expect to be able to do that. No one can. The trick is to be able to break down the problem into smaller and easier ones. The most natural method is to make wishes.

> I have to go through a collection and find elements with a certain property. I wish I had a predicate function that can decide whether an element has that property or not.

There you go! Now you have a simpler problem, just write a predicate function.

Know when to stop

When doing problem seems too difficult even after a long period of time, it is often a good idea to take some rest or do something else. The brain doesn't stop working, it continues subconsciously. So when you revisit the problem, you may just simply see the solution.

Reflection

It is always a good idea to be conscious about the steps we make. This is not to say the obvious that we can't write programs while sleeping. Rather, asking questions like *What did I do? Why did I do that? Is this taking me closer to the goal?* When progress has been made, *Is there a better way? Is this solution can be used for something else?*

We tend do this naturally, but but not to a large extent. It pays off to ask these questions explicitly. Maybe having a checklist of these questions handy when coding is a useful habit.

Do only what is needed

Modern programming languages automate many things. For example, functional languages with `map`, `filter`, `reduce` take care of dealing with collections. You only need to create functions that deal with a single element of the collection.

For people coming from imperative languages, this may go against their instinct first (Where is my for loop?).

Etudes

One can learn the rules of chess or Go immediately. However, mastering them takes long time, playing many games. Same for programming. Learning the basic constructs of a programming language is a quick process. Understanding what `reduce` does is not that hard, but knowing when to use it and how to adapt the general mechanism to a particular problem require lot of practice. Like performing musicians, we need to go through exercises several times.

What is 42?

The task is simple. Write some code that somehow produces 42. The easiest way is to use a data literal `42`. How about other ways? When you are not allowed to write down the number itself. By arithmetic operations,

```
(* 6 7)
(+ 40 2)
```

or by incrementing/decrementing,

```
(inc 41)
(dec 43)
```

or by counting a number of elements in a collection,

```
(count "012345678901234567890123456789012345678901")
```

the ASCII code of `*` is also 42,

```
(int \*)
```

$n!$

Writing a function to calculate $n!$ (n factorial) is a common programming exercise for recursion. It is defined as the product of natural numbers from 1 up to n, with the special case of $0! = 1$. Recursively,

```
(defn factorial [n]
  (if (zero? n)
      1
      (* n (f (dec n)))))
```

but if someone is worried about the inefficiency of the recursive call then tail recursion is also possible.

```
(defn factorial
  ([n] (f n 1))
  ([n r] (if (zero? n)
              r
              (recur (dec n) (* n r)))))
```

The current value to be multiplied with is also passed on through the call, so the caller don't have to wait. However, there is no real need to be recursive, thus checking for the base case is not necessary.

```
(defn factorial [n]
  (reduce * (range 1 (inc n))))
```

Transposing a matrix

Considering matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix},$$

we would like to calculate its transpose (swapping rows with columns)

$$A^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}.$$

Matrices can be represented as nested vectors.

```
(def A [[1 2] [3 4] [5 6]])
```

Without further ado, here is how to do the transpose.

```
(apply map vector A)
([1 3 5] [2 4 6])
```

If the result needs to be a vector as well, then `mapv` can replace `map`.

Bibliography

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. MIT Press, 1996. `https://mitpress.mit.edu/sicp/`, `https://sicpebook.wordpress.com/`, `https://github.com/sarabander/sicp-pdf`.

[2] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979.

[3] D.R. Hofstadter. *Gödel, Escher, Bach. Anniversary Edition: An Eternal Golden Braid*. Basic Books. Basic Books, 1999.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[5] G. Polya and J.H. Conway. *How to Solve It: A New Aspect of Mathematical Method*. Princeton Science Library. Princeton University Press, 2014.

REPLs in the browser

Clojure

REPL.it `repl.it/languages/clojure`

Try Clojure `www.tryclj.com`

ClojureScript

Himera `himera.herokuapp.com`

replumb REPL `clojurescript.io`

klipse `app.klipse.tech`

cljs-webrepl `theasp.github.io/cljs-webrepl`

Solutions to exercises

Solution 1.
`(def x "x")`

Solution 2.
For example, `(zero? (count '(1 2)))`

Solution 3.
It returns the last of the supplied arguments.

Solution 4.
13, since we are hopping through the associations from right to left according to function composition. The number `2` indexes the value `2`, which in turn indexes the keyword `:a` in the middle vector, finally `13` is associated to key `:a`.

Solution 5.
`(defn f [x] (+ (* 4 x x) (* 2 x) -4))`

Solution 6.
`(max a (- a))`, but this only makes the decision invisible, done by `max`.

Solution 7.
`(iterate inc 0)`

Index